IMPORTANT INFORMATION

Please make the following corrections to your Model 4 BASIC Compiler User's Manual:

Page 17, Step 7

```
Type BACKUP :Ø :1 (X) <ENTER>
```

Page 18, Step 4

Enter:

```
*DEMO, TTY=DEMO
```

Pages 65-66

Please note that the source listing format on these pages may not be identical to a BASIC program compiled with the -A switch.

Thank You!
Radio Shack
A Division of Tandy Corporation

Introduction

The BASIC Compiler is an optimizing compiler designed to complement the BASIC Interpreter. The BASIC Compiler lets you create programs that in most cases execute faster than the same interpreted programs, require less memory than the same interpreted programs, and provide source-code security.

These benefits can be critical in the following areas:

- Graphics applications, in which execution speed can often make or break software
- Business applications, in which several CHAINed programs can be supported by a main menu
- Commercial applications, for which software is being sold in a competitive marketplace and source-code security is essential.

The BASIC Compiler supports most of the interpreted BASIC language. Thus, the interpreter and the compiler complement each other, providing an extremely powerful programming environment. You can quickly run and debug a program from within the BASIC Interpreter, and then later compile the program to increase its speed of execution and to decrease its space in memory.

An additional BASIC Compiler feature is BASRUN/CMD, a runtime module that contains most of the runtime environment. The runtime module is loaded when program execution begins, and later execution of CHAINed programs does not require reloading. This lets you develop a system of related programs that can be run using the same runtime environment. And because the runtime environment required by your program need not be saved on disk as part of your executable file, a substantial amount of disk space is saved — typically 45K for a system of four programs.

Although the language supported by the BASIC Compiler is not completely identical to that supported by the BASIC Interpreter, the compiler is designed so that compatibility is maintained wherever possible. The BASIC Compiler supports, in some form, all the statements and commands described in the *BASIC Reference Manual* except:

AUTO
CLOAD
CSAVE
CONT
DELETE
EDIT
ERASE
LIST
LLIST
LOAD
MERGE
NEW
RENUM
SAVE
SOUND

Note: Language, operational, and other differences between the BASIC Compiler and Interpreter are described in Chapter 9 of this manual. Review that information before compiling any of your programs, even those that already run without problem on the interpreter.

System Requirements

The BASIC Compiler requires a TRS-80 Model 4 computer with a minimum of 64K RAM and 2 disk drives. The compiler operates under the TRSDOS 6 operating system.

Royalty Information

For those who want to market application programs, the BASIC Compiler provides three major benefits:

1. Increased speed of execution for most programs
2. Decreased program size for most programs
3. Source code security

When you distribute a BASIC compiled program, you distribute highly optimized machine code, not source code. Consequently, you distribute your program in very compact form and protect your source program from unauthorized alteration.

The policy for distribution of parts of the BASIC Compiler package is as follows:

1. Any application program that you generate by linking to either of the two runtime libraries (BASRUN/REL and BASCOM/REL) may be distributed without payment of royalties. A copyright notice reading "PORTIONS COPYRIGHTED BY MICROSOFT CORPORATION, 1982" must be displayed on the media. Note that programs linked to BASRUN/REL need the runtime module to execute properly.
2. However, the BASRUN/CMD runtime module *cannot* be distributed without first entering into a license agreement with Corporation for such distribution. A copy of the license agreement can be readily obtained by writing to Corporation. Also, a copyright notice reading "PORTIONS COPYRIGHTED BY MICROSOFT CORPORATION, 1982" must be displayed on the media.
3. All other software in your BASIC Compiler package cannot be duplicated except for purposes of backing up your software. Other duplication of any of the software in the BASIC Compiler package is illegal.

All the above information is included in the Nondisclosure Agreement, which must be signed and returned to Microsoft at the time the BASIC Compiler is purchased. In order to provide you any updates or fixes, we must have your completed form on file. Failure to register and sign the Nondisclosure Agreement voids any warranty expressed or implied.

Package Contents

The BASIC Compiler package contains one data disk and one documentation binder.

Software

The BASIC Compiler software contains the following files on disk:

BASCOM/CMD — (the BASIC Compiler) Compiles BASIC source files into relocatable and linkable REL files.

BASRUN/CMD — (the runtime module) A single module containing most of the routines called from your compiled REL file.

BASRUN/REL — (the runtime library) A collection of routines implementing functions of the BASIC language not found in the runtime module. Your REL file may contain calls to these routines.

BASCOM/REL — (the alternate runtime library) A collection of modules containing routines that are similar to the routines found in BASRUN/REL and the runtime module. This library should be used for applications that you want to execute as single executable files without the runtime module. This library does not support CHAIN with COMMON, CLEAR, or RUN <linenumber>. Additional differences are described in Chapter 6, "Linking and Loading."

BCLOAD/L8Ø — (runtime load information) Tells at what address to load your program, and where to find the runtime module at runtime.

DEMO/BAS — (a demonstration program) Used in Chapter 2 to demonstrate program development with the BASIC Compiler.

L8Ø/CMD — (the linking loader) Loads the relocatable REL files into memory, links them into an executable object file, and saves the executable file on your disk.

Documentation

BASIC Compiler User's Manual

The *User's Manual* provides a demonstration run, an introduction to compilation, and a technical reference for use of the BASIC Compiler. It also describes language differences between the BASIC Compiler and the BASIC Interpreter.

Model 4
BASIC Reference Manual

The *Reference Manual* explains syntax and usage of the BASIC language. With the exceptions noted in the *User's Manual*, this is the language supported by the BASIC Compiler.

How To Use This Manual

The *BASIC Compiler User's Manual* is designed for users who are not familiar with the compiler as a programming tool. Therefore, the manual provides a step-by-step introduction to the BASIC Compiler and its use.

This manual assumes a working knowledge of the BASIC language. For reference information, consult the *BASIC Reference Manual.* If you need additional information on BASIC programming, refer to "Learning More about BASIC," below.

INTRODUCTION

Provides brief descriptions of the contents of the BASIC Compiler package and gives a list of references for learning BASIC programming.

Chapter 1 INTRODUCTION TO COMPILATION

Introduces you to the vocabulary associated with compilers, compares interpretation and compilation, and presents an overview of program development with the compiler.

Chapter 2 DEMONSTRATION RUN

Takes you step by step through compiling, linking, and running a demonstration program.

Chapter 3 EDITING A SOURCE PROGRAM

Describes how to create a BASIC source program for later compilation.

Chapter 4 DEBUGGING WITH THE BASIC INTERPRETER

Describes how to debug a BASIC source file with the BASIC Interpreter before compiling it.

Chapter 5 COMPILING

Describes in detail the use of the BASIC Compiler, including command line syntax and compiler options.

Chapter 6 LINKING AND LOADING

Explains how to use L80 to link your programs to needed runtime support.

Chapter 7 RUNNING A PROGRAM

Explains how to run your final executable program.

Chapter 8 METACOMMANDS

Explains how to use the metacommands available with BASIC Compiler.

Chapter 9 A COMPILER/INTERPRETER LANGUAGE COMPARISON

Describes all the language, operational, and other differences between the languages supported by the BASIC Compiler and the BASIC Interpreter. It is important to study these differences and to make the necessary editing changes in your BASIC program before you use the compiler.

The Appendices provide additional technical information and list all BASIC compile time, link time, and runtime error messages.

Syntax Notation

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate data you enter. When the angle brackets enclose lowercase text, you must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, you must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that the you have a choice between two or more entries. You must choose at least one of the entries enclosed in braces unless the entries are also enclosed in square brackets.
|	A vertical bar separates entries within braces. You must choose at least one of the entries separated by bar(s) unless the entries are also enclosed in square brackets.
. . .	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered, exactly as shown.

Learning More About BASIC

The manuals in this package provide complete reference information for your implementation of the BASIC Compiler. They do not, however, teach you how to write programs in BASIC. If you are new to BASIC or need help in learning to program, we suggest you read one of the following books:

Getting Started with TRS-80 BASIC. Radio Shack Catalog #26-2107.

Albrecht, Robert L., Finkel, LeRoy, and Brown, Jerry. *BASIC*. New York: Wiley Interscience, 2d ed., 1978.

Billings, Karen, and Moursund, David. *Are You Computer Literate?* Beaverton, Oregon: dilithium Press, 1979.

Coan, James. *Basic BASIC*. Rochelle Park, N.J.: Hayden Book Company, 1978.

Dwyer, Thomas A., and Critchfield, Margot. *BASIC and the Personal Computer*. Reading, Mass.: Addison Wesley Publishing Company, 1978.

Simon, David E. *BASIC From the Ground Up*. Rochelle Park, N.J.: Hayden Book Company, 1978.

CHAPTER 1

Introduction to Compilation

1.1 Compilation v Interpretation

A microprocessor can execute only its own machine instructions; it cannot execute source program statements directly. Therefore, before you can execute a BASIC program, it must be translated into the machine language of your microprocessor. Compilers and interpreters are two types of translation programs. This chapter explains the difference between these two translation schemes and explains why and when to use the compiler.

Interpretation

BASIC Interpreters (including Disk BASIC and Level II BASIC) translate a BASIC program line by line **at runtime**. To execute a BASIC statement, the interpreter must analyze the statement, check for errors, and then perform the function requested.

If a statement must be executed repeatedly (inside a FOR/NEXT loop, for example), this translation process must be repeated each time the statement is executed.

BASIC programs are stored as a list of numbered lines. During interpretation, a line is not available as an absolute memory address. Therefore, branches such as GOTOs and GOSUBs cause the interpreter to examine all line numbers in the list, starting with the first, until it finds the referenced line.

Similarly, the interpreter maintains a list of all variables. Absolute memory addresses are not associated with the variables in your program. When a statement refers to a variable, the interpreter must search the variables from the beginning until it finds the referenced variable.

Compilation

A compiler translates a source program and creates a new file called an object file. The object file contains "relocatable" machine code, which can be placed and run at different absolute locations in memory. All translation occurs **before** runtime; no translation occurs during execution of the object file. In addition, absolute memory addresses are associated with variables and with the targets of GOTOs and GOSUBs; so lists of variables or of line numbers do not have to be searched during program execution.

The BASIC Compiler is an "optimizing" compiler. Optimizations such as expression reordering and subexpression elimination are made either to increase speed of execution or to decrease program size.

In most cases, BASIC programs execute 3 to 10 times faster than those executed under the interpreter. If you make maximum use of integer variables, execution can be up to 30 times faster. Note, however, that the compiler is not a panacea; you should also examine the algorithms in your programs and the type of processing performed when attempting to increase execution speed.

1.2 Vocabulary

A BASIC program is more commonly called a BASIC "source program" or "source file." The source file is the input file to the compiler and must be in ASCII format. The compiler translates this source and creates, as output, a new file called a "relocatable object" file. The source file has the default extension /BAS, and the relocatable object file has the default extension /REL.

The following terms are stages in the development and execution of a compiled program:

Compiletime — The time during which the compiler is executing and during which it compiles a BASIC source file and creates a relocatable object file.

Link time — The time during which the linker is executing and during which it loads and links together relocatable object files and library files.

Runtime — The time during which a compiled and linked program is executing. By convention, runtime refers to the execution time of your program and **not** to the execution time of the compiler or the linker.

The following terms pertain to the linking process and the runtime support library:

Module — A discrete unit of code. There are several types of modules, including relocatable and executable modules. The compiler creates relocatable modules that the linker can load. Your final executable program is an executable module.

Global Reference — A variable name or label in a given module that is referenced by a routine in another module. Global labels are entry points into modules.

Unbound Global Reference — A global reference in a module that is not declared in that module. The linker tries to resolve this situation by searching for the declaration of that reference in other modules. If it finds such a declaration, it loads that module into memory (if it is not yet in memory), and it becomes part of the load file. These other modules are usually library modules in the runtime library.

If the variable or label is found, the address associated with it is substituted for the reference in the first module and is then said to be "bound." When a variable is not found, it is said to be "undefined."

Relocatable — A module is relocatable if the code within it can be placed and run at different locations in memory. Relocatable modules contain labels and variables represented as offsets relative to the start of the module. These labels and variables are said to be "code relative." When the linker loads the module, an address is associated with the start of the module. The linker then computes an absolute address that is equal to the associated address plus the code relative offset for each label or variable. These new computed values become the absolute addresses that are used in the executable file.

Compiled REL files and library files are all relocatable modules. Normally a relocatable module also contains global references. These are resolved after all local labels and variables have been computed within other relocatable modules. Linking is this process of computing absolute relocated values and resolving global references.

Routine — Executable code residing in a module. More than one routine may reside in a module. The runtime module contains a majority of the library routines needed to implement the BASIC language. A library routine usually corresponds to a feature or subfeature of the BASIC language.

Runtime Support — The body of routines that may be linked to your compiled REL file. These routines implement various features of the BASIC language. Both the runtime libraries and the runtime module contain runtime support routines. See Chapter 6, "Linking and Loading," for more information on runtime support.

The Runtime Module — A module containing most of the routines needed to implement the BASIC language. It is a peculiarity of the runtime module that it is an executable file. The runtime module is named BASRUN/CMD. The runtime module is, for the most part, a library of routines. It is made executable so that you can see the version number of the module.

The BASRUN/REL Runtime Library — A few modules used to load in the runtime module at runtime and to move segments around in memory to permit CHAINing.

The BASCOM/REL Runtime Library — A collection of modules containing routines almost identical in function to similar routines contained in the runtime module and BASRUN/REL.

However, this library does not support the COMMON statement between CHAINed programs. It does support a version of CHAIN that is semantically equivalent to the simple RUN <filename> command.

Linking — The process in which L80 loads modules into memory, computes absolute addresses for labels and variables in relocatable modules, and then resolves all global references by searching the BASRUN/REL runtime library. After loading and linking, the linker saves the modules that it has loaded into memory as a single executable file on your disk.

1.3 The Program Development Process

(The numbers in parentheses refer to Figure 2.1.)

Program development begins with creating (1) a BASIC source file. The best way to create a BASIC source file is with the editing facilities of the BASIC Interpreter, although you can use any general purpose text editor. You must SAVE from the BASIC Interpreter with the **,A** option.

After you write a program, use the BASIC Interpreter to debug the program (2) by running it to check for syntax and program logic errors. There are a few differences in the languages understood by the compiler and the interpreter, but for the most part they are identical. Because of this similarity, running a program provides you with a much quicker syntactic and semantic check of your program than compiling, linking, and finally executing a program. Therefore, you should try to make the interpreter your chief debugging tool.

After you debug your program with the interpreter, compile it (3) to determine differences that may exist between interpreted and compiled BASIC. The compiler flags all syntax errors as it reads your source file. If compilation is successful, the compiler creates a relocatable REL file.

The REL file is not executable and must be linked to one of the runtime libraries (in Figure 2-1 and in the demonstration in Chapter 2, the BASRUN/REL runtime library is used). You may want to include your own assembly language routines to increase the speed of execution of a particular algorithm, or to handle more complex operations. For these cases, use the Editor Assembler to assemble routines (4) that you can later link to your program. Similarly, separately compiled FORTRAN subroutines can be linked to your program.

The linker links all modules (5) needed by your program, and produces as output an executable object file with CMD as the default extension. This file can be executed (6) like any CMD file by simply typing the file's base name (the file name less its CMD extension).

This program development process is demonstrated in the following chapter, Chapter 2, "Demonstration Run."

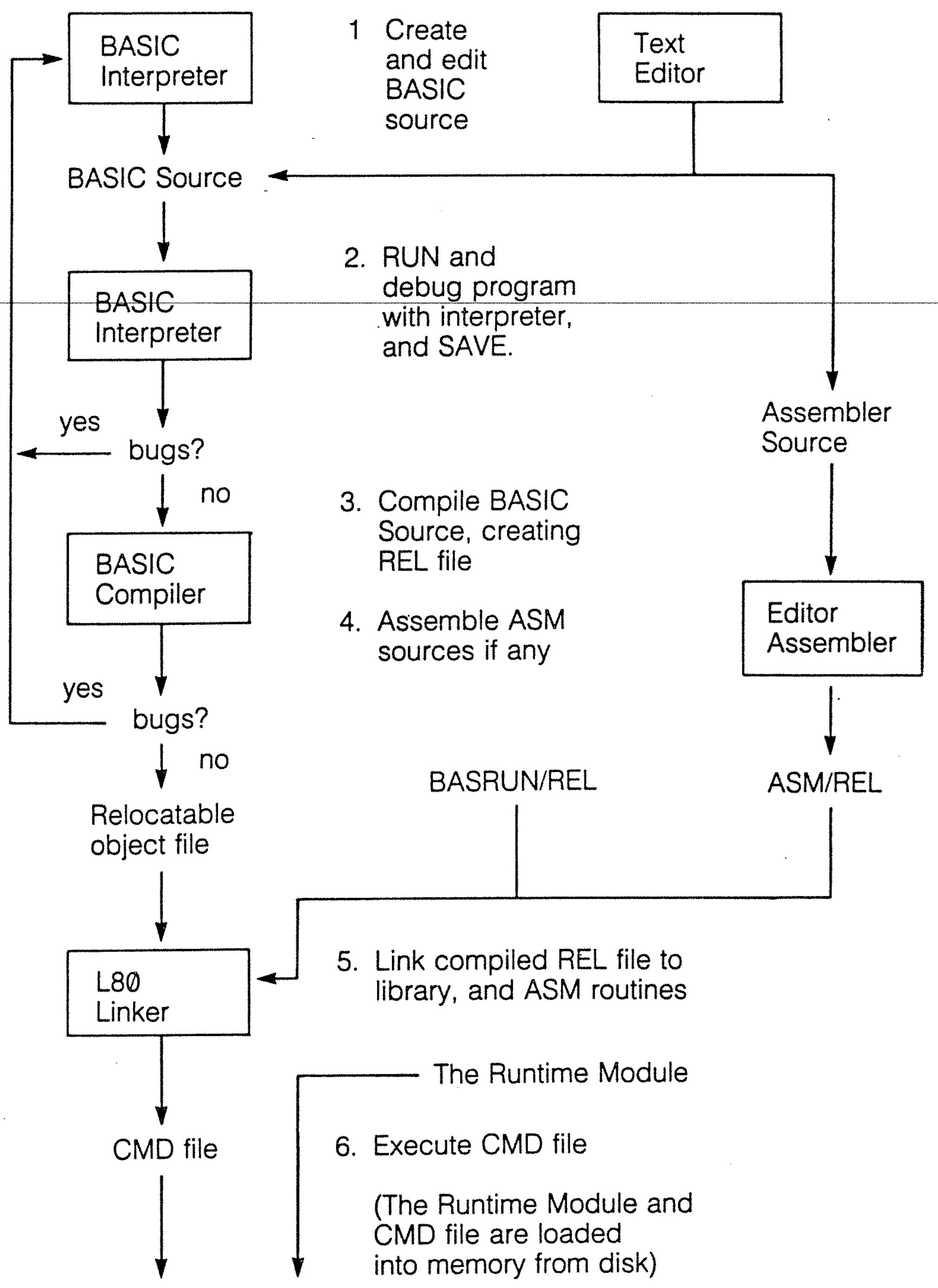
BASIC Interpreter
1 Create and edit BASIC source
Text Editor
BASIC Source
2. RUN and debug program with interpreter, and SAVE.
BASIC Interpreter
yes
bugs?
no
Assembler Source
3. Compile BASIC Source, creating REL file
BASIC Compiler
4. Assemble ASM sources if any
Editor Assembler
yes
bugs?
no
BASRUN/REL
ASM/REL
Relocatable object file
L80 Linker
5. Link compiled REL file to library, and ASM routines
The Runtime Module
CMD file
6. Execute CMD file
(The Runtime Module and CMD file are loaded into memory from disk)

Figure 2.1 The Program Development Process

CHAPTER 2

Demonstration Run

This chapter provides step by step instructions for using the BASIC Compiler. We strongly recommend that you compile the demonstration program in this chapter **before** compiling any other programs. For more technical information, read Chapters 3 through 9.

If you enter commands exactly as described in this chapter, you should have a successful session with the BASIC Compiler. If a problem does arise, carefully redo each step.

Before you begin this demonstration run, back up your BASIC Compiler diskette. Follow the procedure below exactly.

1. Turn on your system as instructed in the *Introduction to Your Disk System* manual.
2. Insert a new, blank diskette in Drive 1 and close the drive door.
3. Insert your TRSDOS system diskette in Drive 0 and close the door.
4. Press the reset button.
5. The screen shows:

   ```
   Date MM/DD/YY ?
   ```

   Enter the date. For example, for July 1, 1984, type **07/01/84.**.
6. When TRSDOS Ready appears, type **FORMAT :1 (Q=N)** (ENTER).
7. When the formatting process is complete, your screen shows:

   ```
   Formatting complete
   TRSDOS Ready
   ```

   Type **BACKUP :0 TO :1X** (ENTER).
8. After the copyright notice appears, the screen shows:

   ```
   Insert SOURCE disk <ENTER>
   ```

   Remove your TRSDOS system diskette, insert your BASIC Compiler diskette in Drive 0, and close the door. Press (ENTER).

   When the backup is complete, the screen shows:

   ```
   Insert SYSTEM disk <ENTER>
   ```

   Remove your backup diskette from Drive 0, insert your system diskette, and press (ENTER).

Check to see if the backup procedure was successful:

1. Remove the original diskette from Drive 1.
2. Place the backup diskette in Drive 1 and close the drive door.
3. Type **DIR :1** (ENTER).

   If the screen displays the directory, your backup was successful.

When you remove the backup diskette from Drive 0, write the diskette name on the label, using a felt-tipped pen.

Store your master diskette in a safe place and always work with the backup copy.

The major stages in developing a program with the BASIC Compiler are:

1. Editing and debugging (entering and correcting the BASIC program, using a BASIC Interpreter)
2. Compiling (creating a relocatable object file)
3. Linking (creating an executable object file)
4. Running (executing the program)

Because we have prepared an edited and debugged demonstration program on disk, you do not have to perform the first two steps described below. Note that we have SAVEd the demonstration program on disk in ASCII format by using the **,A** option. All files must be in ASCII format to be readable by the compiler.

To create an executable compiled program, perform the following steps:

1. Start up your computer system.
2. Create a BASIC source file.

BASIC programs can be created with any available text editor that will create a text file with a logical record length of 1. However, for this demonstration run we will use the program DEMO/BAS, which is provided on your disk. For consistency, BASIC source files should always be given the /BAS extension.

3. To enter the compiler type:

```
BASCOM
```

4. Enter the command line.

After you invoke the compiler, it prints an asterisk to prompt you for the command line. Enter:

```
*DEMO,:TTY=DEMO
```

This command begins compilation of the source file. The source file is the last entry on the command line (DEMO), and the /BAS default extension is assumed.

The compiler generates relocatable object code that is stored in the file specified by the first entry on the command line (DEMO). This file is created with the default /REL extension.

The source listing file is the second entry on the command line. The source listing file is created during compilation. It lists your BASIC source and any compilation errors or warnings as they occur. If no listing file is specified in the command line, none will be generated. For this demonstration, we specified TTY in order to send the source listing file to the console screen.

After you have completed your input, compilation begins. The source listing file is sent to the console screen as the source file is read.

5. Look for error messages.

As your program is compiled, error messages are displayed on the terminal screen. For the demonstration program, there should be no error messages displayed. When the compiler has finished, it displays the message:

```
19329 Bytes Available
18869 Bytes Free

    0 Warning Error(s)
    0 Severe Error(s)
```

(The number of bytes available and bytes free varies with a particular system.) Program control is then returned to the operating system.

At this point, you should see a new file named DEMO/REL listed in the directory.

Note: If your screen shows the error message "Binary Source File," save your program in ASCII format using the -A option.

6. Link routines in the runtime library to your REL file.

Linking is accomplished with the L8Ø linker (the file named L8ØCMD). Perform the following steps to link DEMO/REL to needed runtime support.

a. Invoke L8Ø.

   To invoke L8Ø, enter:

   ```
   L8Ø
   ```

   Your computer will search your disk for L8Ø, load it, and then return the asterisk (*) prompt.

   If you want to stop the linking process, and you have entered only L8Ø and nothing more, you can exit to TRSDOS by pressing the <BREAK> key.

b. Enter the name(s) of file(s) you want loaded and linked.

   L8Ø performs the following operations:

   Loads relocatable object (REL) modules

   Computes absolute addresses for all local references within modules

   Resolves all unbound global references between loaded modules

   Saves the linked and loaded modules as an executable file on disk

   After the asterisk prompt, type the following line to cause loading, linking, and saving of the program DEMO/CMD:

   ```
   *DEMO,DEMO-N-E
   ```

The first part of the command (DEMO) causes loading of the program DEMO/REL. The **–N** switch causes an executable image of the linked file to be saved on your disk with the name DEMO/CMD. This occurs after an automatic search of the BASRUN/REL runtime library. The file is saved only after a **–E** switch is entered on the command line. You may enter as many command lines as needed before you enter a **–E** switch. Note that the **–E** switch causes an EXIT back to TRSDOS. BASRUN/REL is automatically searched to satisfy any unbound global references before linking ends.

c. Wait.

The linking process requires several minutes. During this time, the following messages will appear on your screen:

```
DATA <program-start> <program-
end> <bytes>

<free-bytes> BYTES FREE
[<start-address> <program-end>]
```

This information is described in Chapter 6, "Linking and Loading."

7. Run your program.

To run the executable program, enter:

```
DEMO
```

This causes the runtime module to be loaded. Note that if your system has only one disk drive, the runtime module must be on the disk with DEMO/CMD.

Once the runtime module is loaded, execution of the file named DEMO/CMD begins.

This completes the demonstration run. For more detailed information, see Chapters 3-9 of this manual.

Editing a Source Program

You must write a BASIC source program with a text editor capable of creating a text file with a logical record length of 1. The most convenient is the editor available within the BASIC Interpreter.

The compiler expects its source file in ASCII format. If you edit a file from within BASIC, you must SAVE it with the **,A** option; otherwise, the interpreter encodes the text of your BASIC program into special tokens that the compiler cannot read.

BASIC programs you want to compile are, for the most part, written the same way you write programs to run with the BASIC Interpreter. However, there are certain language differences between the BASIC Interpreter and the BASIC Compiler that must be taken into account when compiling new or existing programs.

One difference is that the compiler supports "metacommands," which are not really part of the BASIC language but rather commands to the compiler itself. The most powerful metacommand is $INCLUDE, which lets you switch BASIC source files during compilation. $INCLUDE and the other metacommands are discussed in Chapter 8.

Another difference is that the interpreter supports a number of editing and file manipulation commands that are useful mainly when creating a program. Examples are LOAD, SAVE, LIST, and EDIT. These are operational commands not supported by the compiler. Some differences also exist for some of the other statements and functions. Remember that the editing stage of program development is when you should account for language differences. See Chapter 9, "A Compiler/Interpreter Language Comparison" for a full description of these differences.

The interpreter cannot accept **physical** lines greater than 254 characters in length. A physical line is the unit of input to the interpreter. Interpreter logical lines can contain as many physical lines as desired.

In contrast to the interpreter, the BASIC Compiler accepts **logical** lines of up to only 253 characters. If you are using an external editor, you can create logical lines containing sequences of physical lines by ending your lines with an underscore. The underscore removes the significance of the carriage return in the <ENTER> sequence that ends each line (underscore characters in quoted strings do not count). This results in just a linefeed being presented to the compiler. The linefeed, <LF>, is the line continuation character understood by the compiler and the interpreter. The ASCII key code for a linefeed is Control-J.

Example:

```
10 IF A=5 THEN PRINT A-
   ELSE PRINT B
```

CHAPTER 3

Debugging with the BASIC Interpreter

The easiest way to debug your BASIC source program is to use the BASIC Interpreter for checking syntax and program logic errors. Debugging with the interpreter is optional; you can create a program without ever running it with the interpreter. You may also edit your program with any general purpose text editor and check for errors at compile time.

You may use some commands or functions in your compiled program that execute differently with the interpreter. In those cases, you need to use the compiler for debugging. Statements supported by the compiler but not by the interpreter are listed in Chapter 9. The compiler also supports double-precision loop control variables and transcendental functions; the interpreter supports neither.

Nevertheless, the language supported by the compiler is intended to be as similar to that of the BASIC Interpreter as possible. This lets you make the BASIC Interpreter your prime debugging tool, saving time by avoiding lengthy compilations and links. Also, the RUN, CONT, and TRON/TROFF statements make the interpreter a very powerful interactive debugging tool. See your *BASIC Reference Manual* for more information on these statements.

The interpreter stops execution of a program when it encounters an error. It does not detect any subsequent errors caught until the first detected error is corrected and the program re-RUN. The compiler scans all lines and reports all detected errors at compile time.

CHAPTER 4

Compiling

After you debug a BASIC source program, your next step is compilation.

5.1 Command Line Syntax

Unlike the BASIC Interpreter, the compiler is not interactive. It accepts only a single command line containing filenames and extensions, appropriate punctuation, optional device designations and switches. How you place these elements when you enter the command line determines which processes the compiler performs. To allow users of single-drive system configurations to use the compiler, the command line can be separated into two command lines: one to invoke the compiler and the other to specify compilation parameters.

BASCOM/CMD may be on a separate disk from the other files. Once loaded, BASCOM/CMD is no longer needed on disk.

The general format for the BASIC Compiler command line is:

```
BASCOM [<objectfile>] [, [<listfile>] ]=<sourcefile>

                         output files        input file
```

where: <**objectfile**> specifies the name to be assigned to the relocatable (REL) object file.

<**listfile**> specifies the name to be assigned to the listing (LST) file.

<**sourcefile**> specifies the name of the BASIC (BAS) source file.

5.2 Using Command Lines

You can specify the following four possible combinations of files on the compiler command line:

1. Relocatable object file (REL) only.
2. Listing file (LST) only.
3. Both a REL and a LST file.
4. Neither a REL file nor a LST file.

5.2.1 Sample Command Lines

Sample command lines are given below for the four possible file combinations.

1. **To Generate an Object File (REL) Only**

   The simplest way to create a REL file is to enter the compiler by typing:

   ```
   BASCOM <objectfile>=<sourcefile>
   ```

   <objectfile> defaults to the default drive (normally :0). This

CHAPTER 5

may or may not be the disk on which <sourcefile> resides. You may give an optional device designation to either <objectfile> or <sourcefile>.

2. **To Generate a Listing File (LST) Only**

   To create only a listing file, enter the compiler by typing:

   ```
   BASCOM ,<listfile>=<sourcefile>
   ```

   The generated <listfile> contains a line-by-line listing of the BASIC source. If you use the **-A** compiler switch described later in this section, the object code generated for each BASIC statement is disassembled and listed along with the corresponding BASIC statements in your program. The actual REL file is not in a human-readable form.

   To print out a listing file, enter the command line with the name of the line printer device (LPT) in place of the listing filename:

   ```
   BASCOM ,LPT=<sourcefile>
   ```

   When you examine your listing, notice the two hexadecimal numbers preceding each line of the source program. The first number is the relative address of the code associated with that line, using 0 as the start of the program. The second number is the cumulative data area needed so far during the compilation. These two columns are totaled at the end of the listing. The left column total is the actual size of the generated REL file in bytes. The right column total is the total data area required in bytes.

3. **To Generate both Object and Listing Files**

   To generate **both** object and listing files, enter the compiler by typing:

   ```
   BASCOM <objectfile>,<listfile>=
   <sourcefile>
   ```

   The <objectfile> and <listfile> parameters default to the default drive. You may add optional device designations at the end of these parameters.

   When your compilation is finished, the following message is displayed:

   ```
   xxxxx Bytes Available
   xxxxx Bytes Free

   xxxxx Warning Error(s)
   xxxxx Severe Error(s)
   ```

   If severe errors occur, correct them and recompile the program.

4. **To Suppress Generation of All Output Files**

   To perform a syntax check of your <sourcefile>, while suppressing generation of either an <objectfile> or a

<listfile>, enter the compiler by typing:

```
BASCOM =<sourcefile>
```

or

```
BASCOM ,=<sourcefile>
```

In this example, the compiler simply compiles the source program and reports the number of errors and the number of free bytes. This is the fastest way to perform a syntax check of your program with the compiler. Running a program with the interpreter lets you perform an accurate syntax check only insofar as the language of the BASIC Interpreter supports the language of the BASIC Compiler.

You may want to create output files on a disk other than the defaults provided by the compiler, or you may want to create output files with different extensions or base names than that of your BASIC source file. To do so, you must actually specify the filenames with the desired extensions or device designations, as described in the following sections.

5.2.2 Filename Extensions

You may append a filename extension of up to three characters to a filename. These extensions may contain any alphanumeric character, given in any position. Lowercase letters are converted to uppercase. Extensions must be preceded by a slash (/).

The BASIC Compiler and L80 recognize certain extensions by default. If you give your filenames unique extensions, you must always remember to include the extension as part of the filename for any filename parameter. When you omit filename extensions, the Compiler assumes default extensions.

The TRSDOS default filename extensions used with the compiler are:

Extension	**Type of File**
/BAS	BASIC source file
/REL	Relocatable object file
/CMD	Executable object file
/LST	Listing file
/MAC	Editor Assembler source file

5.2.3 Device Designations

Each command line field may include device designations that instruct the compiler where to find files or where to place them.

The disk drive designation is placed after a filename. For example:

```
DEMO:1
```

For the input file (the sourcefile), the drive designation indicates from which device the file is read. For output files (objectfile or listfile), the drive designation indicates where the files are written.

Device names supported under TRSDOS are:

Designation	**Device**
:0, :1, :2, :3	Disk Drives
:LPT	Line Printer
:TTY	Console

When you omit device names, the default is drive :0 if the file does ~~not exist; otherwise it is the lowest drive number containing a file~~ with the same name.

For example, the following command line:

```
BASCOM =DEMO:1
```

directs the compiler to write the object file to drive :0 if DEMO/REL does not exist. The following output files are written to drive :0, if :0 is the currently logged drive:

```
BASCOM DEMO,DEMO=DEMO
BASCOM ,DEMO=DEMO
```

When the compiler has finished, it exits to TRSDOS and the currently logged drive.

5.2.4 Device Names As Filenames

One command line option is to give a device name in place of a filename. The result of this option depends on which device you specify, and for which command line parameter. Figure 6.1 illustrates some possibilities:

DEVICE	**<objectfile>**	**<listfile>**	**<sourcefile>**
:0, :1, :2, :3	writes file to drive specified	writes file to drive specified	N/A (must be specified in its entirety)
LPT	N/A (unreadable file format)	writes listing to line printer	N/A (output only)
TTY	N/A (unreadable file format)	sends listing to console	

N/A = Not Allowed

Figure 6.1 Effects of Using Device Designations in Place of File Names

The BASIC Compiler is not an interactive program. However, use of device names in place of filenames lets you compile lines input directly from the keyboard, display lines on the screen as they are compiled, or print out lines on a printer as they are compiled. For example:

,TTY (Console) may be entered in place of list filename.

Example:

```
DEMO,TTY=DEMO
```

displays the list file (source and compiled code) for each line on the screen as it is compiled.

,LPT (Lineprinter) may be entered in place of list filename.

Example:

```
DEMO,LPT=DEMO
```

prints the list file (source and compiled code) for each line on the line printer as it is compiled.

5.3 Compiler Switches

In addition to specifying filenames, extensions, and devices to direct the compiler to produce object and listing files, you can direct the BASIC Compiler to perform additional or alternate functions by adding switches to the command line.

You can place switches after source filenames or after other switches, as in the following command line:

```
BASCOM FOO,FOO=FOO-D-X
```

Switches signal special instructions to be used during compilation. The switch tells the compiler to "switch on" a special function or to alter a normal compiler function. You may use more than one switch, but all must begin with a hyphen (–). **Do not confuse these switches with the linker switches**, which are discussed in Chapter 6.

The three types of compiler switches are convention, error-handling, and special code.

Convention switches let you specify which of two lexical (language) and execution conventions you want applied during compilation: version 4.51 or version 5.0. Version 5.0 is the default. If your programs contain Version 4.51 features, use the **-4** switch.

Error-handling switches let you compile source programs that contain error-handling routines involving the ON ERROR GOTO statement plus some form of a RESUME statement. The two error-handling switches are **-E** and **-X**. Error-handling routines require line numbers in the REL file. If you do not use one of the error-handling switches, the compiler does not place line numbers in the REL file. Thus, if a RESUME statement or ON ERROR GOTO statement is encountered, a severe compiler error results.

Special code switches cause the compiler to generate special code for certain uses or situations. Some of these special code switches cause the compiler to generate larger and slower code.

Figure 6-2 summarizes the functions of these switches. Following the figure are detailed descriptions of each switch.

CATEGORY	SWITCH	ACTION
Conventions	-4	Use 4.51 lexical conventions.
	-T	Use 4.51 execution conventions.
	-N	Relax line-numbering constraints (not allowed with -4).
	-5	Use BASIC Version 5.3 convention (default). Use -4-5 together for BASIC lexical but Version 5.3 execution conventions. Use -T-5 together for BASIC execution but Version 5.3 lexical conventions.
Error-Handling	-E	Program has ON ERROR GOTO with RESUME <line number>.
	-X	Program has ON ERROR GOTO with RESUME, RESUME 0, or RESUME NEXT.
Special Code	-Z	Use Z80 instructions (default).
	-I	Use only 8080 compatible instructions in the compiled code.
	-A	Include listing of disassembled object code in the listing file.
	-O	Substitute the BASCOM/REL runtime library for BASRUN/REL as the default runtime library searched by the linker.
	-D	Generate debug code for runtime error checking.
	-S	Write quoted strings to REL file on disk and not to data area in RAM.

Figure 6-2. Compiler Switches

5.3.1 Convention Switches

The default convention switch (**-5**) provides Version 5.0 lexical and execution conventions. The **-4-T** switch may be used for Version 4.51 conventions.

Switch	**Action**
-4	Directs the compiler to use the lexical conventions of the version 4.51 BASIC Interpreter. Lexical conventions are the rules that the compiler uses to recognize the BASIC language.

The following conventions are observed:

1. Spaces are not significant.
2. Variables with embedded reserved words are illegal.
3. Variable names are restricted to two significant characters.

The **-4** switch forces correct compilation of a source program in which spaces do not delimit reserved words, as in the following statement.

```
FORI=ATOBSTEPC
```

With the **-5** switch, the compiler assigns the variable "ATOBSTEPC" to the variable "FORI" With the **-4** switch set, the compiler recognizes the line as a FOR statement.

Note: The **-4** and **-N** switches may not be used together.

-T	Tells the compiler to use BASIC version 4.51 execution conventions. Execution conventions refer to the implementation of BASIC functions and commands and what they actually do at runtime.

With **-T** specified, the following 4.51 execution conventions are used:

1. FOR/NEXT loops are always executed at least one time.
2. TAB, SPC, POS, and LPOS perform according to version 4.51 conventions.
3. Automatic floating-point to integer conversions use truncation instead of rounding, except when a floating-point number is being converted to an integer in an INPUT statement.
4. The INPUT statement leaves the variables in the input list unchanged if only a carriage return is entered. If a "?Redo from start" message is issued, then a valid input list must be given. A carriage return in this case generates another "?Redo from start" message.

-5 Tells the compiler to use BASIC Version 5.3 execution and lexical conventions. This switch is the default convention.

1. Variable names may be a maximum of 40 characters, with all 40 characters significant. You may use letters, numbers, and the decimal point in variable names, but the name must begin with a letter.

   Variable names may also include embedded reserved words. Reserved words include all BASIC commands, statements, function names, and operator names.
2. FOR . . . NEXT loops are not executed if the final value is less than the initial value and the increment is positive.
3. The INPUT statement changes the variables in the input list to the null string or zero if you only press (ENTER).

-N Tells the compiler to relax line numbering constraints. When you specify **-N**, line numbers in your source file may be in any order, or they may be eliminated entirely. With **-N**, lines are compiled normally, but unnumbered lines cannot be targets for GOTOs or GOSUBs. While **-N** is set, the underline character causes the remainder of the physical line to be ignored. In addition, **-N** causes the underline character to act as a line feed so that the next physical line becomes a continuation of the current logical line. (See Chapter 3 for more information on physical and logical lines.)

The **-N** switch provides three advantages:

1. Elimination of line numbers increases program readability.
2. The BASIC Compiler optimizes over entire blocks of code rather than single lines (for example in FOR . . . NEXT loops.)
3. BASIC source code can more easily be included in a file with $INCLUDE.

Remember that -N and -4 may not be used together.

5.3.2 Error-Handling Switches

The error-handling switches let you use ON ERROR GOTO statements in your program. These statements can aid you greatly in debugging your BASIC programs. However, extra code is generated by the compiler to handle ON ERROR GOTO statements.

Switch	Action
-E	Tells the compiler that the program contains an ON ERROR GOTO/RESUME <line number> construction. To handle ON ERROR GOTO properly, the compiler must generate extra code for the GOSUB and RETURN statements. Yoy must include a line number address table (one entry per line number) in the REL file so that each runtime error message includes the number of the line in which the error occurs. To save memory space and execution time, do not use this switch unless your program contains an ON ERROR GOTO statement. **Note:** The only RESUME statement that works properly with **-E** is RESUME <line number>. If your program uses RESUME, RESUME NEXT, or RESUME 0 with an ON ERROR GOTO statement, use the **-X** switch instead.
-X	Tells the BASIC Compiler that the program contains one or more RESUME, RESUME NEXT, or RESUME 0 statements. The **-X** switch performs all the functions of the **-E** switch, so you never need to use the two at the same time. For instance, the **-X** switch, like the **-E** switch, causes a line number address table to be included in your object file so that each runtime error message includes the number of the line in which the error occurs. With **-X**, however, the line number address table contains one entry per statement; with **-E**, the table contains one entry per line number. In order that RESUME statements may be handled properly, the compiler cannot optimize across statements. Therefore, do not use **-X** unless your program contains RESUME statements **other than** RESUME <line number>.

5.3.3 Special Code Switches

The special code switches are:

Switch	Action
-Z	Tells the compiler to use Z80 instructions whenever possible. This switch is the default mode. When the **-Z** switch is set, several additional Z80 instructions are allowed in addition to the 8080 compatible instructions normally generated by the compiler.
-I	Tells the compiler to use 8080 compatible instructions, rather than Z80 instructions. The object code is still listed using Z80 instructions, however.
-A	Includes the disassembled object code for each source line in the source listing file.

-O Tells the compiler to substitute the BASCOM/REL runtime library for BASRUN/REL as the default runtime library searched by the linker. When you use this switch, you cannot use the runtime module.

Note: CMD files created by linking to BASCOM/REL do not need the runtime module on disk at runtime.

-D Generates debugging and error handling code at runtime. Use of **-D** lets you use TRON and TROFF in the compiled file. Without **-D** set, TRON and TROFF are ignored.

With **-D**, the BASIC Compiler generates somewhat larger and slower code that performs the following checks:

1. **Arithmetic overflow**. All arithmetic operations, both integer and floating-point, are checked for overflow and underflow.
2. **Array bounds**. All array references are checked to see if the subscripts are within the bounds specified in the DIM statement.
3. **Line numbers**. The generated binary code includes line numbers so that the runtime error listing can indicate the line on which an error occurs.
4. **RETURN**. Each RETURN statement is checked for a prior GOSUB statement.

If the **-D** switch is not set, array bound errors, RETURN without GOSUB errors, and arithmetic overflow errors do not generate error messages at compile time. At runtime, no error messages are generated either, and erroneous program execution may result. Use the **-D** switch to make sure that you have thoroughly debugged your program.

-S The **-S** switch forces the compiler to write quoted strings that are longer than 4 characters to your REL file on disk as they are encountered, rather than retaining them in memory during the compilation of your program. If this switch is not set, and your program contains a large number of long quoted strings, you may run out of memory at compile time.

Although the **-S** switch allows programs with many quoted strings to take up less memory at compile time, it may **increase** the amount of memory needed in the runtime environment, since multiple instances of identical strings will exist in your program. Without **-S**, references to multiple identical strings are combined so that only one instance of the string is necessary in your final compiled program.

CHAPTER 6

Linking and Loading

A linking loader performs two important programming functions. First, it loads into memory one or more program files you select. The files that L8Ø loads are called REL files. REL files are created during the compilation process and contain relocatable machine code. A REL file is not an executable file. Converting a REL file into an executable object file, a process known as linking, is the second function of the linking loader. Specifically, the linking loader (L8Ø) searches the REL file (or files, if more than one has been loaded) for all references to subroutines needed to perform BASIC or other functions such as floating point addition, printing data, and so on.

Some of the subroutines that are needed are in BASRUN/CMD, the runtime module, which will be brought into memory just prior to execution of your program. Others, the less commonly used subroutines, are in BASRUN/REL, the subroutine library. For each BASIC function, there is either a complete subroutine (or series of subroutines) stored in BASRUN/REL or there is a reference to a subroutine stored in the runtime module.

L8Ø searches BASRUN/REL to satisfy undefined globals. If the subroutine needed is stored in BASRUN/REL, L8Ø links that subroutine to the loaded program(s). If the required subroutine is stored in the runtime module, L8Ø sets up the code necessary for the program to find the subroutine.

The final action of the linking loader, if the programmer requests it, is to save the loaded program(s) and the linked routines in a single executable disk file. This file is automatically given the extension /CMD, unless you specify otherwise.

In addition to these basic link/loading functions, L8Ø can load and link assembly language subroutines written with the Editor Assembler or FORTRAN Compiler, both available as separate products. L8Ø also lets you control where program and data areas are placed.

6.1 Linker Command Lines

A simple linker command line might look like this on your screen:

```
L8Ø

*PROG/CMD-N,PROG/REL-E
```

The asterisk (*) is the L8Ø prompt. PROG/CMD is the name of the executable file to be created; it is followed by the names of the REL files to be linked. **Note:** Linker switches have no relation to the compiler switches discussed in the preceding chapter.

If you want default filename extensions, you need not include them in the command line:

```
L8Ø

*PROG-N,PROG-E
```

You can type both parts of the command line on the same line. For example, the following command performs the same functions as the preceding example:

```
L8Ø PROG-N,PROG-E
```

In any of the above examples, the -N switch indicates that an executable file is to be created, and the -E switch tells the linker to exit to TRSDOS and store the executable file on disk. Before exiting, the linker automatically searches BASRUN/REL on the currently logged drive for any as yet undefined global references. (To search BASCOM/REL instead, use the -O compiler switch; see section 6.2.) The final linked executable file has the name specified by your <filename>-N command. **Note:** The -N switch is essential if you want to create an executable file.

You **must** specify the name of the file to store on disk. If you do not, no executable file is stored.

Linker switches are discussed in detail in Section 6.4.

To link an assembly language subroutine to your BASIC program, you can type, for example:

```
L8Ø

*PROG,MYASM,PROG-N-E
```

In this case, MYASM/REL is the name of the assembly language subroutine, and PROG/REL is the name of your program. The subroutine MYASM/REL **cannot** be assembled with an END <label> statement. The linker assumes that <label> is the start address of a separate **program**, and the linker refuses to link two programs together because their two separate start addresses will conflict.

When you link a REL file to BASRUN/REL, the BCLOAD/L8Ø file must be on disk in the currently logged drive. If it is not, your screen displays the following error message:

```
?BCLOAD not found, please create
header file
```

More information about BCLOAD/L8Ø is in Section 6.3.

When your linking session is complete, your screen displays the following message:

```
DATA <program-start> <program-end>
<bytes>

<free-bytes> BYTES FREE
[<start-address> <program-end>]
```

1. <program-start> is the hexadecimal address of the beginning of your program.
2. <program-end> is the hexadecimal address of the end of your program.

3. <bytes> is the decimal size of your program in bytes.
4. <free-bytes> is the decimal size of unused memory in bytes during linking.
5. <start-address> is the hexadecimal start address of your program (not necessarily the same as <program-start>).

Parameters 1, 2, 3, and 5 are referenced by number in Figure 6.1, which shows the link data map for a program linked to BASRUN/REL and using the runtime module. If you link to BASCOM/REL and use the -P and -D linker switches, some of this information is not accurate (see Section 6.4 for details on linker switches).

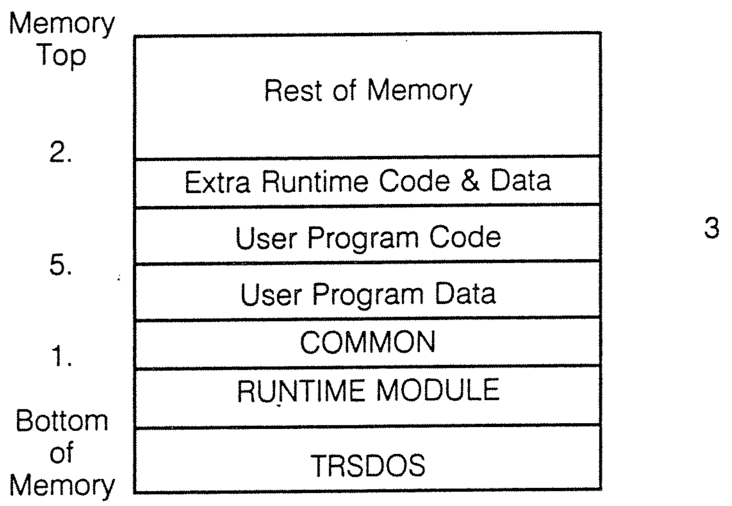

Figure 6.1 Link Data Map

For programs linked to BASRUN/REL and using the runtime module, the size of the executable program (CMD file) in bytes is roughly equal to:

<program-end> – <start-address>*1.01

Remember that at runtime the runtime module resides in memory along with your executable file. When execution of your program begins, the first step is to load the runtime module to establish the runtime support environment.

6.2 Runtime Support

Once you compile a REL file, you must link your program to modules that contain runtime support routines. Runtime support is the body of routines that, in essence, implement the BASIC language. Your compiled REL file, on the other hand, implements the particular algorithm that makes your program a unique BASIC program.

Runtime support is essential to the execution of all compiled BASIC programs. It is found in the runtime module and the runtime library. As a rule, only a portion of all possible runtime routines is linked to your REL file.

The time required for linking all the necessary runtime support routines is often a problem on microcomputers. Partly for this reason, the runtime module contains all of the more frequently used routines in one module. Since they all reside in one module, they are all linked at once and need not be searched for in later linker searches. The runtime module is automatically linked to every program via a dummy module in BASRUN/REL; it is not present in memory at linktime. (Thus, any program is at least 16K long at runtime.) If your program needs other less frequently used routines, these routines are automatically searched for and found in BASRUN/REL. At linktime, you **cannot** use the -P and -D linker switches, since they will cause errors at runtime. **Note:** The runtime module must be accessible on disk when the CMD file is executed.

When you specify the -O switch at **compiletime**, the alternate runtime library (BASCOM/REL) is substituted for BASRUN/REL as the default library to be searched at **linktime**. At linktime you can then use -P and -D as described in Section 6.4. **Note:** When BASCOM/REL is selected as the library to be searched, the runtime module is not used by your program at all.

There are some advantages to using the BASCOM/REL runtime:

1. For small, simple programs you may be able to compile and link programs smaller than the 21K minimum required to accommodate the BASRUN/CMD module. This can be important in compiling a program for a ROM-based application, where space is a critical factor.
2. Execution of a compiled and linked CMD file does not require that the runtime module be on disk at runtime.

There are, however, some distinct advantages to using the BASRUN/CMD runtime module:

1. You can use COMMON and CHAIN statements to support a system of programs sharing common data. With BASCOM/REL, COMMON is not supported and CHAIN is semantically equivalent to RUN.
2. With BASRUN/CMD the CLEAR command is implemented; it is **not** implemented with BASCOM/REL.
3. The RUN <linenumber> option to RUN is implemented with BASRUN/CMD; it is **not** implemented with BASCOM/REL.
4. When BASRUN/CMD is used, the linker can load programs approximately 12K larger than when BASCOM/REL is used. In addition, linktime is reduced, since unbound globals do not have to be searched for in multiple library modules.
5. The routines in BASRUN/CMD are not incorporated into the executable file. This can save approximately 16K of disk space per executable file; it also results in slightly faster CHAINing.

For more information on using CHAIN and COMMON with a system of programs, see Appendix A.

6.3 The BCLOAD File

Because of the way the BASIC runtime environment is implemented with the runtime module, the file BCLOAD/L8Ø must be on one of the disk drives at linktime.

BCLOAD/L8Ø contains two pieces of information: the hexadecimal load address of your program, and the filename of the runtime module.

BCLOAD/L8Ø looks like this if you LIST it out:

```
+83ØØ       [program load address]
BASRUN/CMD  [filename of the
            runtime module]
```

At runtime, you must have the runtime module in the disk drive specified in the BCLOAD/L8Ø file, or an error is generated. The default location of the runtime module is the currently logged drive. With any available text editor, you can alter BCLOAD/L8Ø, before **linktime**, to specify the disk on which you want the runtime module to reside at **runtime**.

The plus sign (+) tells the linker to write the CMD file beginning at the start address of your program instead of at the program load address. (The start address is the address at which your program begins execution.)

6.4 Linker Switches

As with the BASIC Compiler, you can use switches with L8Ø to specify certain functions. Unlike compiler switches, however, L8Ø command line switches are not always placed at the end of the command line. Most are placed at the end of the command line, but some must be placed at the beginning, and some in the middle.

Table 6.1 lists the switches available with L8Ø. **Do not confuse these switches with the compiler switches.**

Table 6.1. Linker Switches

CATEGORY	SWITCH	ACTION
Exit	-E	Exit to TRSDOS
Save	-N	Save all previously loaded programs and subroutines using the name immediately preceding -N
	-N:P	Alternate form of -N; save only program area
Address Setting	-P	Set start address for programs and data. If used with -D, -P sets only the program start
	-D	Set start address for data area only
	-R	Reset L8Ø
Library Search	-S	Search the library named immediately preceding -S
Global Listing	-U	List undefined globals and program and data area information (a direct command)
	-M	List complete global reference map
Radix Setting	-O	Octal radix
	-H	Reset to hexadecimal radix (default)

Two switches are used in **every** linking session. These are the switches in the first two categories — Exit and Save.

6.4.1 Exit Switches

Switch	**Action**
-E	Causes L8Ø to execute and then causes program control to Exit from L8Ø and return to TRSDOS command level. Every link loading command line should end with the -E switch. Although it is possible to exit L8Ø in other ways (press <BREAK> when at L8Ø command level), the -N switch has no effect until L8Ø sees the -E switch.

6.4.2 Save Switch

Switch	**Action**
-N	Saves a memory image of the executable file on disk, using the filename and extension you specify. If you do not specify an extension, the default extension for the saved file is /CMD. Unless the command line contains this switch, no memory image of the linked file is saved on disk. Therefore, you use this switch almost

every time you link a REL file. To specify which drive contains the diskette for saving the memory image, insert the drive number (:d) between the filename and -N.

The -N switch must immediately follow the filename of each file you wish to save, and it does not take effect unless a -E switch follows it. Once the file is saved on disk, you need only type BASRUN filename at TRSDOS command level to run the program.

The default condition of saving an executable file is to save both program and data areas. If you wish to save only the program to make your disk files smaller, use the -N switch in the form -N:P. With this switch set, only the program code is saved. Do not use this -N:P feature if you compiled your program with the -S switch.

These two switches are all that are required in most L80 operations. Some additional functions are available through the use of other switches that let you manipulate the L80 processes in more detail. The switches that turn on these additional functions are arranged in categories according to type of function. The function of each category is defined by the category name.

6.4.3 Address Setting Switches

Switch	Action
-P	Sets both the program and data origin. The format of the -P switch is -P:<address> The address value must be expressed in the current radix. The default radix is hexadecimal. You will know if the radix is set for a base other than hexadecimal because the radix can only be changed by giving a switch in the L80 command line. The default value for the -P switch is :3000. You may use this switch to set program and data origin higher to make room for small machine language subroutines, as described in Section D.2.
-D	Sets the data area origin by itself. Since the program origin always starts exactly at the end of the data area, unless otherwise specified, the -D switch used by itself has the exact same effect as the -P switch used by itself. The syntax for the -D switch is the same as for the -P switch: -D:<address> The address for the -D switch must be in the current radix. (Hexadecimal is the default radix.)

When you use the -P switch with the -D switch, data areas load starting at the address given with the -D switch. (The program loads beginning at the program origin given with the -P switch.) This is the **only** occasion when the address given in -P: is the start address for the actual program code.

The -D switch, like the -P switch, takes effect as soon as L80 "sees" the switch (the effect is not deferred until linking is finished), but the -D switch has no effect on programs already loaded. Therefore, it is important to place the -D switch (as well as the -P switch) **before** the data (and programs) you want to load at the address specified.

You must separate the -P switch and -D switch from the REL filename with a comma. For example,

```
L80 -P:5600,DEMO,DEMO-N-E
```

Additional Note for -P and -D Switches

If your program was compiled with the -O switch and is too large for the linking loader, you may be able to load it anyway if you use -D and -P together. This way you can load programs and data of approximately 27K to 30K, depending on the number of global symbols.

While L80 is loading and linking, it builds a table consisting of five bytes for each program relative reference. If you use the -D, -E, or -X switch, this table contains at least five bytes for every line number. By setting both -D and -P, you eliminate the need for L80 to build this table, thus giving you some extra memory.

The -D and -P switches should not be used for programs using the runtime module.

To set the two switches, look at the end of the LPT file listing. Take the number for the total of data, add that number to 3000H, add another 100H+1, and the result should be the -P: address for the start of the program area. The -D switch should be set to -D:3000.

-R	Resets L80 to its initialized condition. L80 scans the command line before it begins the functions commanded. As soon as L80 sees the -R switch, all files loaded are ignored, L80 resets itself, and the asterisk (*) prompt is returned, showing that L80 is running and waiting for you to enter a command line. The version of L80 supplied with the BASIC Compiler defaults the initial load address to 3000H. The default save file extension is CMD.

6.4.4 Library Search Switch

Switch	Action
-S	Causes L80 to search the file named immediately prior to the switch for routines, subroutines, definitions for

globals, and so on. In a command line, the filename with the -S switch appended must be separated from the rest of the command line by commas.

The -S switch is used to search library files only, such as BASCOM or FORLIB.

You rarely need to give the -S switch. Only under the following conditions is it required:

1. Use BASCOM-S if you have only one drive (see steps for Running L8Ø given above).
2. Use FORLIB-S to search the FORTRAN runtime library if one or more of the programs you are link loading is a FORTRAN program.

```
Example:  SAMPLE/FOR -N
          SAMPLE/REL, FORLIB-S
```

The extensions are optional.

6.4.5 Global Listing Switches

Switch	**Action**
-U	Tells L8Ø to list all undefined globals, to the point in the link session when L8Ø encounters the -U switch. Although this switch is not a default setting, if your program contains any undefined globals, they are listed automatically, just as if you had set the -U switch. If your program contains no undefined globals, the actions controlled by the -U switch do not occur unless -U is given in a command line that does not end with a -E switch. Globals are the names of assembly language subroutines that are called from the REL file. If L8Ø cannot find the routine, the global is undefined. Unless you have written some of your own subroutines and have directed L8Ø to load and link them with your compiled program, you should have no need to use this switch. BASRUN provides definitions for the globals you need to run your program. In addition to listing undefined globals, the -U switch directs L8Ø to list the origin and the end of the program and data areas. However, the program portion of the information is listed only if the -D switch was also given. If the -D switch was not given also, the program is stored in the data area, and the origin and end of the data area include the origin and end of the program.
-M	Displays (lists) all globals, both defined and undefined, on the screen. The listing cannot be sent to a printer. In the listing, defined globals are followed by their values, and undefined globals are followed by an asterisk (*).

Both the -M switch and the -U switch list the program and data area information.

6.4.6 Radix Setting Switches

Switch	Action
-O	Sets the current radix to Octal. If you have a reason to use octal values in your program, give the -O switch in the command line.
-H	Resets the current radix to Hexadecimal. Hexadecimal is the default radix. You do not need to give this switch in the command line unless you previously gave the -O switch and now want to return to hexadecimal.

CHAPTER 7

Running a Program

To run a compiled program, simply enter the filename without its CMD filename extension. For example:

```
DEMO
```

This command causes execution of the program DEMO/CMD. If the program has been linked with the BASRUN/REL runtime library, the runtime module must be accessible from disk at runtime.

The executable binary file can also be executed from within a program, as in the following statement:

```
10 RUN "PROG"
```

The default extension is /CMD. The CMD file can be a binary file created in any programming language. The CHAIN command is used in a similar fashion. In either case, an executable binary file is loaded. The runtime module is not reloaded when you use CHAIN; it is when you use RUN.

It is important to realize that the bulk of the runtime environment is taken up by the runtime module. This module is automatically loaded when you initially invoke an executable file requiring the runtime module. When you RUN a program, the executable file is loaded into memory. The runtime module is also loaded to create a fresh runtime environment. Both files reside in memory simultaneously.

CHAPTER 8

Metacommands

Metacommands are compiler directives that provide two capabilities: source file control and listing file control. The available metacommands are listed in Table 8.1.

Table 8.1 The Metacommands

Name	Default (+/-)	Description
$INCLUDE:'<filename>'		Switches compilation from current source file to source given by <filename>.
$LIST+		Turns on or off source listing. Errors are always listed.
$OCODE+		Turns on or off disassembled object code listing.
$TITLE:'<text>'		Sets page title.
$SUBTITLE:'<text>'		Sets page subtitle.
$LINESIZE:n		Sets width of listing. Default is 80.
$PAGESIZE:n		Sets length of listing in lines. Default is 66; 60 are printable.
$PAGE		Skips to next page. Line number is reset.
$PAGEIF:n		Skips to next page if less than (n) lines left.
$SKIP:n		Skips (n) lines or to end of page.

+ = on
- = off

8.1 Syntax

You can give one or more metacommands at the start of a comment. Multiple metacommands are separated by whitespace characters: space, tab, or linefeed. Whitespace between the elements of a metacommand is ignored. Therefore, the following metacommands are equivalent:

```
REM $PAGE:12
REM $PAGE : 12
```

Note: Do not space between the dollar sign and the rest of the metacommand.

To disable metacommands within comments, place a character that is not a tab or space before the **first** dollar sign. For example:

```
REM x$PAGE:12
```

Except for $INCLUDE, the metacommands affect the source listing only. Many commands can be turned on and off within a listing. For example, most of a program might use $OCODE-, with a few sections using $OCODE+ as needed. However, some metacommands, because of their nature, apply to an entire compilation.

In the metacommands listed in Section 8.2, the following rules apply:

1. A metacommand followed by plus (+) or minus (-) is an on/off switch.
2. The plus (+) or minus (-) given in the heading for each description of an on/off switch is the default setting of the given metacommand.
3. A metacommand followed by :n requires an integer (Ø < n < 256).
4. A metacommand followed by :'<text>' requires a string.

8.2 Descriptions

The metacommands available with the BASIC Compiler are:

8.2.1 $INCLUDE:'<filename>'

Lets the compiler switch processing of a source file from the current source to the BASIC file given by the <filename> parameter. When the end of file is reached in the included source, the compiler switches back to the original source and continues compilation. Resumption of compilation in the original source file begins with the line of source text that follows the line in which the $INCLUDE occurred. Therefore, REM $INCLUDE should always be the last statement on a line, since the remainder of the line is always treated as part of a comment.

$INCLUDEd BASIC source files may be subroutines, single lines, or any type of partial program. **Note:** You must enclose <filename> in single quotes; the default extension is /BAS.

Be sure that any variables in the included files match their counterparts in the main program and that included lines do not contain GOTOs to nonexistent lines, END statements, or similarly erroneous code.

These further restrictions must be observed:

1. You must SAVE included files with the ,A option if they were created from within the BASIC Interpreter.
2. Included lines must be in ascending order.
3. The lowest line number of the included lines must be higher than the line number of the $INCLUDE metacommand in the main program.
4. The range of line numbers in the included file must numerically precede subsequent line numbers in the main program.

   These restrictions do not apply if the main program is compiled with the -N switch set, since line numbers need not be in ascending order in this case. For more information, see Section 5.3, "Compiler Switches."

5. You cannot nest $INCLUDE metacommands inside other include files. This means that you can use $INCLUDE only in the file containing your main BASIC program; a $INCLUDE metacommand **cannot** appear inside the included source file.

6. The $INCLUDE directive must be the last statement on a line and must be part of a comment statement, as in the following statement:

```
999 DEFINT I-N : REM $INCLUDE:"COMMON/BAS"
```

All other metacommands are designed to control the source listing. Note, however, that none of the metacommands listed below have any effect if NUL/LPT is the name of the source listing file.

8.2.2 $LIST+

Turns on the source listing; $LIST- turns it off. Metacommands themselves appear in the listing, except for $LIST-. The format of the listing file is described in Appendix B, "Listing File Format."

8.2.3 $OCODE+

Controls listing of the generated code in the listing file. For each BASIC source line, code addresses and operation mnemonics are listed. Note that $OCODE- turns off listing of the generated code, even if the -A switch is used when you enter the compiler. $OCODE+ turns on the generated code listing, regardless of the use of -A.

8.2.4 $TITLE:'<text>'

Sets the name of a title that appears at the top of each page of the source listing. The string <text> must be fewer than 60 characters.

8.2.5 $SUBTITLE:'<text>'

Sets the name of a subtitle that appears beneath the title at the top of each page of the source listing. The string <text> must be fewer than 60 characters.

8.2.6 $LINESIZE:n

Sets the maximum length of lines in the listing file. This value defaults to 80. The number of characters printed per line is (n - 1). The integer n must be greater than 40.

8.2.7 $PAGESIZE:n

Sets the maximum size of a page in the source listing. The default is 66. To allow space for the page header, a page has (n-6) lines printed on it. The integer n must be greater than 16.

8.2.8 $PAGE

Forces a new page in the source listing. The page number of the listing file is automatically incremented.

8.2.9 $PAGEIF:n

Conditionally performs $PAGE, above, if there are fewer than n printed lines left on the page. If n or more lines are left on the page, no action is taken.

8.2.10 $SKIP:n

Skips n lines in the source listing file. If fewer than n lines are left on the current page, the listing skips to the start of the next page.

CHAPTER 9

A Compiler/Interpreter Language Comparison

You must be aware of the differences between the languages supported by the BASIC Compiler and the BASIC Interpreter when compiling existing or new BASIC programs. For this reason we recommend that you first compile the demonstration program in Chapter 2, then read Chapters 3-9, and only then begin compiling other programs.

The differences fall into three main categories: operational, language, and others. The lists on the next page serve as a reference guide to these differences, with detailed discussion following.

All commands and functions except the metacommands and commands specific to BASIC are also described in the *BASIC Reference Manual*. We suggest that for a complete understanding of a command or function, you read the information in the *BASIC Reference Manual*, then see this chapter for specific differences between the interpreter and compiler implementations.

9.1 Operational Differences

BASIC Interpreter operational commands are not acceptable as input to the compiler. These include:

AUTO
CLOAD
CONT
CSAVE
DELETE
EDIT
ERASE
LIST
LLIST
LOAD
MERGE
NEW
RENUM
SAVE
SOUND

9.2 Language Differences

Most programs that run under the BASIC Interpreter compile under the BASIC Compiler with little or no change. However, it is necessary to note differences in the following commands:

CALL
CHAIN
CLEAR
COMMON
DEFxxx
DIM
END
FOR/NEXT

FRE
MEM
ON ERROR GOTO
REM
RESUME
RETURN
RUN
STOP
TRON/TROFF
USR
WHILE/WEND
WIDTH

The differences in the interpreter/compiler implementations of these commands and statements are described below.

9.2.1 CALL

The CALL statement lets you call and transfer program control to a precompiled FORTRAN subroutine or to an assembly language routine created with the Editor Assembler.

The format of the CALL statement is:

CALL <global-name> [(<argument-list>) . . .]

<global-name> is the name of the subroutine you wish to call. This name must be 1 to 6 characters long and must be recognized by L8Ø as a global symbol. That is, <global-name> must be the name of the subroutine in a FORTRAN SUBROUTINE statement or a PUBLIC symbol in an assembly language routine.

<argument-list> is optional. It contains arguments that are passed to an assembly language or FORTRAN subroutine.

Note: It is the responsibility of the assembly language procedure to preserve the values in the registers at the point where the procedure was invoked.

Further information on assembly language subroutines is in the discussion of the USR function that follows. For more information on creating and interfacing assembly language routines, see Appendix D.

Example:

```
    120 CALL MYSUBR (I,J,K)
```

Note: If you do not have FORTRAN, you can only use the CALL statement with assembly language subroutines.

9.2.2 CHAIN

The BASIC Compiler does not support the ALL, MERGE, DELETE, and <line number> options to CHAIN. If you wish to pass variables, we recommend that you use the COMMON statement.
Note: Files are left open during CHAINing.

The default filename extension is /CMD. BASIC compiler programs can chain to any CMD file; however, they do not pass any command line information.

See Appendix A for examples of programs using CHAIN.

9.2.3 CLEAR

The BASIC Compiler supports the CLEAR command. The format is:

```
CLEAR [,<expression1> [,<expression2> ] ]
```

> <expression1> and <expression2> must be integer expressions. If specified, the first expression sets the highest memory location available at compile time. If specified, the second expression sets the number of bytes available for the stack during compilation.
>
> If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 256 bytes, and the default top of memory is the current top of memory.

Note: CLEAR is supported only for programs using the runtime module and not for programs linked to the BASCOM/REL runtime library.

The CLEAR statement performs the following actions:

Closes all files
Clears all COMMON and user variables
Resets the stack and string space
Resets all numeric variables and arrays to zero
Resets all string variables and arrays to null
Releases all disk buffers

See Appendix C for a memory map showing the location of the stack, string space, and disk buffers discussed above.

The compiler's CLEAR statement does **not** clear DEFxxx statements, as does the interpreter's. For the compiler, these declarations are fixed at compiletime and may not vary.

9.2.4 COMMON

The BASIC Compiler supports a modified version of the COMMON statement. COMMON must appear in a program before any executable statement. All statements are executable **except**:

COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
$Metacommands

You must declare arrays in COMMON in preceding DIM statements, and array names must have parentheses when used in COMMON. For example:

```
COMMON A ( ) ,B$ ( ) ,C ( )
```

The standard form of the COMMON statement is referred to as "blank" COMMON. FORTRAN-style "named" COMMON areas are also supported; however, the named COMMON variables are not preserved across CHAINs.

The format for named COMMON is:

COMMON /<name>/ <list of variables>

where: <name> is 1 to 6 alphanumeric characters, starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to explicitly pass parameters in the CALL statement.

Blank COMMON is used for passing variables between programs. It is named blank COMMON because COMMON regions are not specified. For blank COMMON statements communicating between CHAINing and CHAINed-to programs, the order of the variables must be the same in both programs if the sizes of blank COMMON are different.

To ensure that COMMON areas can be shared between programs, place blank COMMON declarations in a single include file and use the $INCLUDE statement in each program. For example:

```
MENU/BAS

    10 ' $INCLUDE: 'COMDEF'
     .
     .
     .
    1000 CHAIN "PROG1"

PROG1/BAS

    10 ' $INCLUDE: 'COMDEF'
     .
     .
     .
    2000 CHAIN "MENU"

COMDEF/BAS

    100 DIM A(100),B$(200)
    110 COMMON I,J,K,A()
    120 COMMON A$,B$(),X,Y,Z
    130 REM END COMDEF/BAS
```

Note: COMMON is not supported by the BASCOM/REL runtime library. Therefore, do not compile programs with the -O switch if they contain COMMON statements.

9.2.5 DEFINT/SNG/DBL/STR

DEFxxx statements designate the storage class and data type of variables listed as parameters.

The compiler does not "execute" DEFxxx statements, as it does PRINT statements, for example. A DEFxxx statement takes effect as soon as it is encountered in your program **during compilation**. Once the type is defined for the listed variables, that type remains in effect until the end of the program or until another DEFxxx statement alters the type of the variable. Unlike the interpreter, the compiler cannot circumvent the DEFxxx statement by directing flow of control around it with a GOTO. For variables given with a precision designator (i.e., %, !, #, as in A% = B), the type is not affected by the DEFxxx statement.

At compile time, the compiler allocates memory for storage of designated variables and assigns them one of the following data types:

> **INT**eger,
> **S**i**NG**le precision floating-point,
> **D**ou**BL**e precision floating-point, or
> **STR**ing.

9.2.6 DIM

The DIM statement is similar to the DEFxxx statement in that it is scanned rather than executed. That is, DIM takes effect when it is encountered at compile time and remains in effect until the end of the program; it cannot be reexecuted at runtime.

If the default dimension (10) is already established for an array variable, and that variable is later encountered in a DIM statement, an "Array Already Dimensioned" error results. Therefore, the practice of putting a collection of DIM statements in a subroutine at the end of your program generates severe errors. In that case, the compiler sees the DIM statement only after it has already assigned the default dimension to arrays declared earlier in the program.

The values of the subscripts in a DIM statement must be integer constants; they may not be variables, arithmetic expressions, or floating-point values.

9.2.7 END

During execution of a compiled program, an END statement closes files and returns control to the operating system. The compiler assumes an END statement at the end of the program, so "running off the end" (omitting an END statement at the end of the program) produces proper program termination by default.

9.2.8 FOR/NEXT

You can use double precision FOR/NEXT loops with the compiler. All FOR/NEXT loops must be statically nested ; that is, each FOR must have a single corresponding NEXT. Static nesting also means that each FOR/NEXT pair must reside within an outer FOR/NEXT pair. Therefore, the following construction is **not** allowed:

```
10  FOR I=1 TO 10
20     FOR J=1 TO 10
30        FOR K=1 TO 10
 .
 .
 .
70     NEXT J
80        NEXT K
90  NEXT I
```

The following form is correct:

```
10  FOR I=1 TO 10
20     FOR J=1 TO 10
30        FOR K=1 TO 10
 .
 .
 .
70        NEXT K
80     NEXT J
90  NEXT I
```

In addition, do **not** direct program flow into a FOR/NEXT loop with a GOTO statement. The result of such a jump is undefined, as in the following example:

```
50  GOTO 100
 .
 .
 .
90  FOR I=1 TO 10
 .
 .
 .
100 PRINT "INLOOP"
 .
 .
 .
200 NEXT I
```

9.2.9 FRE

The compiler supports two versions of the FRE statement, one with a numeric argument and the other with a string argument.

Examples:

Y = FRE (X)
Y = FRE (S$)

FRE with a numeric argument always returns zero (0).

FRE with a string argument causes string space to be compacted so that the free string space is not fragmented. Then FRE returns size of this single block of string space.

9.2.10 MEM

In the BASIC Interpreter, MEM returns the amount of free space in memory. In the compiler, it always returns zero (Ø).

9.2.11 ON ERROR GOTO

If a program contains ON ERROR GOTO and RESUME <linenumber> statements, you must include the -E compilation switch in the compiler command line. If you use the RESUME, RESUME NEXT, or RESUME Ø form, you must use the -X switch instead.

The purpose of these switches is to allow the compiler to function correctly when error-handling routines are included in a program. See Section 5.3, "Compiler Switches," for a detailed explanation of these switches. **Note:** Using these switches increases the size of the REL and CMD files.

9.2.12 REM

REM statements are REMarks starting with a single quotation mark or the word REM. Since REM statements do not take up time or space during execution, you may use REM as desired. This practice improves the readability of your programs.

9.2.13 RESUME

See the preceding discussion of ON ERROR GOTO.

9.2.14 RETURN

In addition to the simple RETURN statement, the compiler supports RETURN <linenumber>. This allows a RETURN from a GOSUB to an arbitrary <linenumber>, thereby circumventing normal return of program control to the statement following the GOSUB statement.

9.2.15 RUN

The BASIC Compiler supports both RUN and RUN <linenumber>. It does not support the "R" option with RUN. If you desire this feature, use the CHAIN statement.

Note: RUN is used to execute CMD files created by the BASIC Compiler and does not support the execution of BASIC source files, as does the interpreter. Other CMD files not created with the BASIC Compiler **are** executable with the RUN statement. These can be CMD files created in languages other than BASIC.

9.2.16 STOP

The STOP statement is identical to the END statement, except that it terminates your program at a point that is not necessarily its end. It also prints a message telling you at which hexadecimal address you stopped. If the -D, -E, or -X compiler switches are turned on, then the message prints the line number at which you stopped. As with the END statement, STOP closes all open files and returns control to the operating system. STOP is normally used for debugging.

9.2.17 TRON/TROFF

To use TRON/TROFF, the compiler -D **D**ebug switch must be used. Otherwise, TRON and TROFF are ignored and a warning message is generated.

9.2.18 USR

Although the USR function is implemented in the compiler to call machine language subroutines, you can only pass parameters through the use of POKEs to protected language routine. See Appendix D for details on using the USR function.

9.2.19 WHILE/WEND

Like FOR/NEXT loops, WHILE/WEND constructions must be statically nested. Static nesting means that each WHILE/WEND pair, when nested within other FOR/NEXT or WHILE/WEND pairs, cannot reside partly in and partly outside the nesting pair. For example, the following construction is **not** allowed:

```
FOR I=1 to 10
    A=COUNT
    WHILE  A=1
NEXT I
    A=A-1
    WEND
```

In addition, do not direct program flow into a WHILE/WEND loop without entering through the WHILE statement.

See "FOR/NEXT," Section 9.2.8, for an example of this restriction and for an example of correct static nesting.

9.2.20 WIDTH

The WIDTH statement sets the printed line width in number of characters for the terminal or line printer.

The format is:

```
WIDTH [LPRINT] <integer expression>
```

If you omit the LPRINT option, the line width is set at the terminal. If you include LPRINT, the line width is set at the line printer.

The <integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If <integer expression> is 255, the line width is "infinite"; that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

9.3 TRS-80 Commands in BASIC

The BASIC Compiler supports the following TRS-80 commands:

CLS
MEM
PRINT@
RANDOM
TIME$
Graphics BASIC Commands (requires graphics board)
SYSTEM ["command"]

9.4 BASIC Compiler Features not in BASIC Interpreter

The BASIC Compiler supports some powerful and efficient features not supported by BASIC Interpreter. These new features compile with no problems, but keep in mind that you cannot run a program using these features with your interpreter.

1. Double Precision Transcendental Functions

   SIN, COS, TAN, SQR, LOG, and EXP return double-precision results if given a double-precision argument. Exponentiation with double-precision operands returns a double precision result.

2. Fixed Stack

   The BASIC Compiler uses a 256-byte fixed stack at the top of memory. Consequently, you cannot branch indefinitely. For every GOSUB issued, the program must execute a RETURN. Nesting is allowed, but only up to 100 levels. If this limit is not observed, your program crashes.

9.5 Other Differences

Other differences between the BASIC Interpreter and the BASIC Compiler include the following:

1. Expression Evaluation — The BASIC Compiler performs optimizations, if possible, when evaluating expressions.
2. Use of Integer Variables — The BASIC Compiler can make optimum use of integer variables as loop control variables. This allows some functions (and programs) to execute up to 30 times faster than when interpreted.
3. Double Precision Arithmetic Functions — The BASIC Compiler allows double-precision arithmetic functions, including all the transcendental functions.
4. Double Precision Loop Variables — Unlike the interpreter, the BASIC Compiler allows the use of double-precision loop control variables.

5. String Space Implementation — To increase the speed of garbage collection, the implementation of the string space for the compiler differs from its implementation for the interpreter.

9.5.1 Expression Evaluation

During expression evaluation, the BASIC Compiler converts operands of different types to the type of the more precise operand. For example, the following expression:

```
QR=J%+A!+Q#
```

converts J% to single precision, adds it to A!, converts the result to double-precision, and adds it to Q#.

The BASIC Compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter, the following statements yield 20000 for A%.

```
K%=20000
I%=20000
J%=20000
A%=(I%+J%)-K%
```

That is, J% is added to I%. Because the resulting number is too large for an integer representation, the interpreter converts the result into a floating-point number. The final result (20000) is found and converted back to an integer and saved as A%.

The BASIC Compiler, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode and arithmetic overflow occurs. If the -D **D**ebug switch is set, the error is detected. Otherwise, an incorrect answer is produced.

Besides the above type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

```
I%=20000
J%=-18000
K%=20000
M%=I%+J%+K%
```

If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, overflow does occur. The compiler follows the rules of operator precedence, and you may use parentheses to direct the order of evaluation. **You can guarantee evaluation order in no other way.**

9.5.2 Integer Variables

To produce the fastest and most compact object code possible, make maximum use of integer variables. For example, the following program executes approximately 30 times faster by replacing "I", the loop control variable, with "I%" or by declaring I an integer variable with DEFINT.

```
FOR I=1 TO 10
A(I)=0
NEXT I
```

Also, it is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

9.5.3 Double Precision Arithmetic Functions

The BASIC Compiler lets you use double-precision floating-point numbers as operands for arithmetic functions, including all the transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, SQR). The interpreter supports only single-precision arithmetic functions.

Your program development strategy when designing a program with double precision arithmetic functions should be as follows:

1. Implement your BASIC program using single-precision operands for all functions that you later intend to be double precision.
2. Debug your program with the interpreter to determine the soundness of your algorithm before converting variables to double precision.
3. Declare all desired variables as double precision. Your algorithm should be sound at this point.
4. Compile and link your program. It should implement the algorithm that you have already debugged with the interpreter, but with double the precision in your arithmetic functions.

9.5.4 Double Precision Loop Control Variables

The compiler, unlike the interpreter, lets you use double precision loop control variables. You may, therefore, increase the precision of increment in loops.

9.5.5 String Space Implementation

The compiler and interpreter differ in their implementations and maintenance of string space. With the compiler, using either PEEK or POKE with VARPTR, or using assembly language routines to change string descriptors, may result in a "String Space Corrupt" error. See more information on string space in the discussion of the CALL statement, Section 9.2.1, and the USR function, Section 9.2.17.

APPENDIX A

Creating a System of Programs with the Runtime Module

The CHAINing with COMMON feature and the runtime module are designed for creating large systems of BASIC programs that interact with each other. In this section, a hypothetical system is described to show the interactions in a large system.

The following integrated accounting system contains separate packages for general ledger, accounts payable, and accounts receivable. Within each package, the components are separate programs, each of which is separately compiled, linked, and loaded. At linktime, the load address in the BCLOAD file should be the same for all programs.

A main menu program controls entry into each package. The system structure is shown below:

```
                       MENU


        GL                   AP                  AR


  GL01 GL02 GL03     AP01  AP02  AP03     AR01 AR02 AR03
```

To use CHAINing with COMMON effectively, you must logically structure the system and the COMMON information. In the system pictured above, COMMON information exists within each of the packages GL, AP, and AR. Each package contains a system of three separately compiled programs. Furthermore, there may be COMMON information between MENU and each of the packages. Note that there may be overlapping sets of COMMON information.

The compiler's COMMON statement is not as flexible as the interpreter's: With the compiler, COMMON areas must be the same size in programs that CHAIN to each other.

This may be accomplished by:

1. Using the same COMMON declarations in all programs so that all common information may be shared.
2. Using the same set of COMMON declarations within each of the three packages, with no information shared with the other packages or the main MENU program via COMMON. In this case, there will be three sets of COMMON declarations, one for each package.

For a large, integrated system of compiled programs the second method gives more flexibility with the compiler, because program control is switched from package to package through the main MENU. Any common information that could be obtained from MENU is obtained instead from the main program for each of the packages GL, AP, and AR. This approach would be used with a single package.

For the system shown above, the use of CHAIN and RUN commands in each major program is outlined in the following program fragments. **Note:** RUN loads the specified program as a normal executable file and starts execution. For compiled BASIC programs, a new copy of the runtime module is reloaded at that time, allowing a new system of CHAINed programs to be started. While CHAINing is in progress, the runtime module is in control and therefore does not have to be reloaded for each program.

```
MENU/BAS

          1000 IF MENU=1 THEN RUN "GL"
          1010 IF MENU=2 THEN RUN "AP"
          1020 IF MENU=3 THEN RUN "AR"

GL/BAS                          General Ledger

          10 ' $INCLUDE: 'GLCOMDEF'
                 (GL) COMMON declarations
          1000 CHAIN "GL01"
          1010 CHAIN "GL02"
          1020 CHAIN "GL03"
          1030 IF MENU=YES THEN RUN "MENU"

AP/BAS                          Accounts Payable

          10 ' $INCLUDE: 'APCOMDEF'
                 (AP) COMMON declarations
          1000 CHAIN "AP01"
          1010 CHAIN "AP02"
          1020 CHAIN "AP03"
          1030 IF MENU=YES THEN RUN "MENU"

AR/BAS                        Accounts Receivable

          10 ' $INCLUDE: 'ARCOMDEF'
                 (AR) COMMON declarations
          1000 CHAIN "AR01"
          1010 CHAIN "AR02"
          1020 CHAIN "AR03"
          1030 IF MENU=YES THEN RUN "MENU"
```

Each of the lower level programs XXYY (XX=GL, AP, AR, YY = 01, 02, 03) should CHAIN back to the package main program XX.

APPENDIX B

Source Listing Format

The source listing file format is described below. The discussion is keyed to the sample listing on the next page.

Every page of the source listing has a header at the top. The left portion of the first two lines contains the user-assigned title and subtitle, which are set with the first source line.

In some versions of BASIC, the right side of the second line contains the date, and the right side of the third line contains the time. The "Offset" column specifies the hexadecimal offset from the start of the executable file for each line of source. The "Data" column specifies the hexadecimal offset from the start of the data segment for any data values generated by the source line. The "Source Line" column contains a source line's line number, along with the line itself. This line number and the source file name identify runtime errors if the appropriate error checking is on.

Example: The following source listing is from a program compiled with the -A switch (to include listing of disassembled object code).

```
BASIC Compiler                                          PAGE 1

Program

 Offset  Data  Source Line              BASIC Compiler V5.34

 0014   0007    10 ' $TITLE: 'BASIC Compiler' $SUBTITLE:
'Program'
 0014   0007    20 DEFINT A-Z
 0014   0007    30 DIM A(50)
 0014   0007    40 ' $OCODE +
 0014   0007    50 A(0) = 1: A(1) = 1
 0014    **          I00000: CALL    $432
 0017    **          L00010: L00020: L00030: L00040:
L00050:
 0017    **                         LD     HL,0001
 001A    **                         LD     (A %),HL
 001D    **                         LD     (A %+0002),HL
 0020   006D    60  FOR I=1 TO 24
 0020    **             L00060:     LD     HL,0001
 0023    **                         LD     (I %),HL
 0026    **              I00001:
 0026   006F    70   A(2*(I+1)) =
A(2*(I+1)-1)+A(2*(I+1)-2)+3
 0026    **              L00070:    LD     HL,(I %)
 0029    **                         ADD    HL,HL
 002A    **                         ADD    HL,HL
 002B    **                         PUSH   HL
 002C    **                         LD     DE,A %+0002
 002F    **                         ADD    HL,DE
 0030    **                         LD     E,(HL)
 0031    **                         INC    HL
 0032    **                         LD     D,(HL)
 0033    **                         EX     DE,HL
 0034    **                         LD     (T:01),HL
```

```
0037    **                      POP    HL
0038    **                      PUSH   HL
0039    **                      LD     DE,A %
003C    **                      ADD    HL,DE
003D    **                      LD     E,(HL)
003E    \\                      INC    HL
003F    \\                      LD     D,(HL)
0040    **                      LD     HL,(T:01)
0043    **                      ADD    HL,DE
0044    **                      INC    HL
0045    **                      INC    HL
0046    **                      INC    HL
0047    **                      LD     (T:02),HL
004A    **                      POP    HL
004B    **                      LD     DE,A %+0004
004E    **                      ADD    HL,DE
004F    **                      PUSH   HL
0050    **                      LD     HL,(T:02)
0053    **                      EX     DE,HL
0054    **                      POP    HL
0055    **                      LD     (HL),E
0056    **                      INC    HL
0057    **                      LD     (HL),D
0058    006F    80   NEXT I
0058    **              L00080: LD     HL,(I %)
005B    **                      INC    HL
005C    **                      LD     (I %),HL
005F    **                      LD     HL,(I %)
```

Two kinds of compiler messages appear in the listing: severe errors and warnings. Do not link a compilation with severe errors. You can use one with only warnings to generate code, but the result may not execute correctly. Errors and warnings are listed in Appendix G, "Error Messages."

Usually the location of an error in the source line is indicated by a caret (^), followed by a two-character code. At times, however, an error in a line is not immediately detected, and the error indicator may point to the end of a statement or the end of a line. This is normally the case with TC ("too complex") errors.

APPENDIX C

The following memory maps show layout of the runtime memory for programs linked to the two runtime libraries, BASRUN/REL and BASCOM/REL. Remember if you link to BASRUN/REL, the runtime module is used at runtime.

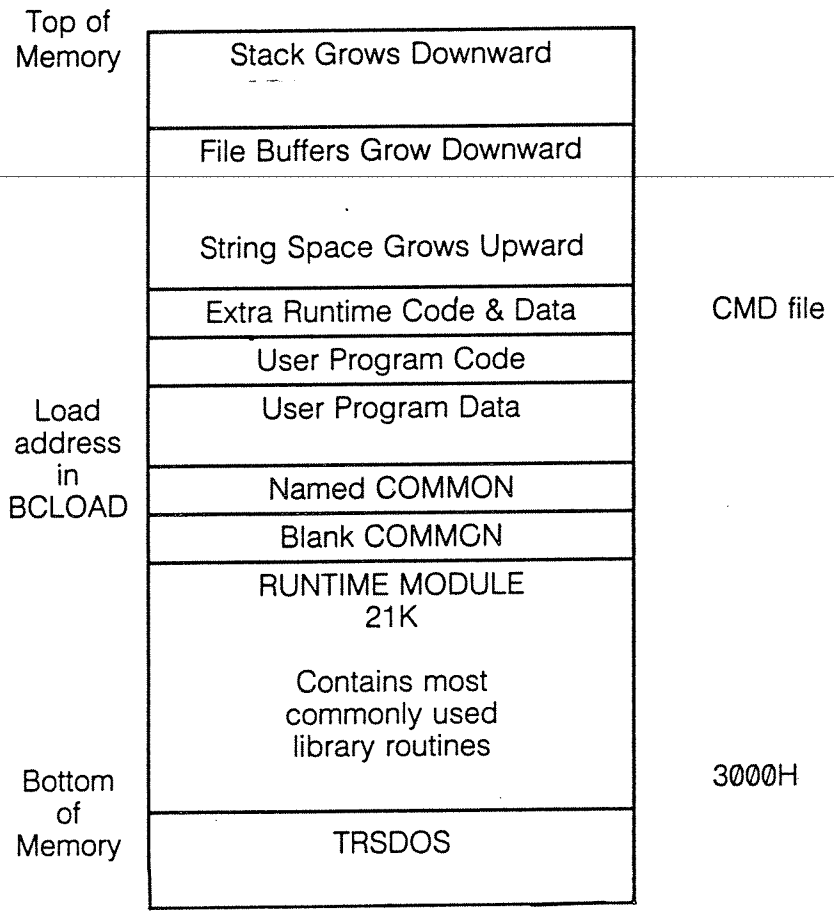

Figure C-1. Runtime memory map of a program using the BASRUN/CMD runtime module.

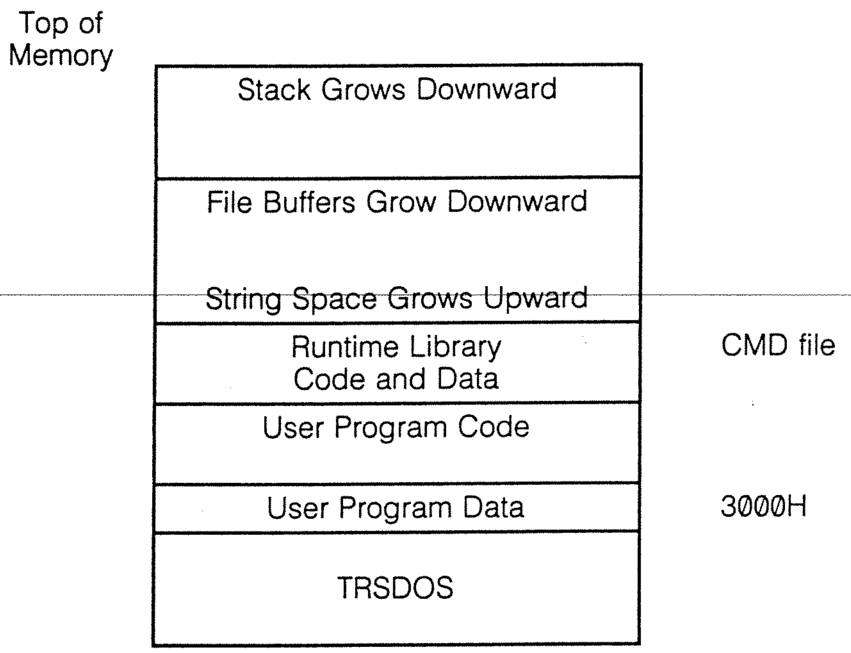

Figure C-2. Runtime memory map of a program using the BASCOM/REL runtime library.

APPENDIX D

BASIC Assembly Language Subroutines

All versions of BASIC have provisions for interfacing with assembly language subroutines via the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way BASIC intrinsic functions are called. However, the CALL statement is the recommended way of interfacing 8080/Z80 machine language programs with BASIC. It is compatible with more languages than is the USR function call, it produces more readable source code, and it can pass multiple arguments.

D.1 Memory Allocation

It is important to avoid stack overflow when calling assembly language subroutines. Therefore, if additional stack space is needed when an assembly language subroutine is to be called, the BASIC stack can be saved and a new stack set up for use by the assembly language subroutine. The BASIC stack must be restored, however, before returning from the subroutine.

You can load the assembly language subroutine into memory by means of the operating system or the BASIC POKE statement. If you have the Editor Assembler, routines can be assembled with the Editor Assembler and loaded using L80.

D.2 USR Function Calls

Although the CALL statement is the recommended way of calling assembly language subroutines, the USR function call is still available for compatibility with previously written programs.

The syntax of the USR function is:

```
    USR[ <digit> ] [ (<argument> ) ]
```

where: <digit> is from 0 to 9. <digit> specifies which USR routine is being called (see DEF USR statement in the *BASIC Reference Manual*). If <digit> is omitted, USR0 is assumed.

<argument> is ignored by the compiler. Arguments may be passed only with the use of POKE statements to memory locations known by the assembly language procedure (see discussion below).

For each USR function, a corresponding DEF USR statement must have been executed to define the USR call offset. This offset determines the starting address of the subroutine.

When the compiler sees X = USRn (0), it generates the following code:

```
    CALL        $U%+const
    LD          (X%),HL
```

During execution, the program encounters this code, jumps to the address of the CALL, performs the steps of your subroutine, and then returns to resume execution where it left off. Your routine should place the integer result of the routine in the HL register pair prior to returning to the compiled BASIC program.

On return, as shown above, the contents of the HL register pair are placed in the location of the variable X. Any other parameters to be passed must be PEEKed from the main BASIC program, and POKEd into protected memory locations. With this method of passing parameters, the USR function works quite well. You must take responsibility, however, to ensure that your code and any variables you use are protected. This is more complicated than in the interpreter because the top of memory pointer cannot be set from within the compiled program. It must be set prior to executing the compiled program, if any part of high memory is to be protected.

If you do not want to use the above method of passing parameters, you have three other choices:

1. If your machine language routine is short enough, you can store it by making the first string defined in the program contain the ASCII values corresponding to the hexadecimal values of your routine. Use the CHR$ function to insert ASCII values in the string. You can then find the start of your routine by using the VARPTR function.

   For example, for the string A$, VARPTR (A$) returns the address of the length of the string. The next two addresses are (first) the least significant byte and (then) the most significant byte of the actual address of the string. This set-up of the string space is different from the interpreter's.

   Thus, to find the actual start address of your routine, use the following BASIC instructions:

   ```
   A$ = "String containing routine"
   I% = VARPTR (A$)
   AD% = PEEK (I% + 2) * 256 + PEEK (I% + 1)
   ```

   AD% is the start address of your routine.

Note: Since strings move around in the string space, you must adjust absolute references to reflect the current memory location of the routine. To make your code position independent for the Z80, use relative rather than absolute jumps.

2. The second method is to reset the default value of the load address in the BCLOAD/L80 file. The BCLOAD/L80 file's main purpose is to direct loading of your executable program so that later the runtime module can be loaded beneath it in memory. By increasing the load address, you create free protected space between the end of the runtime module and the start of the loading area. If you increase the load address by 100H, for example, 256 bytes of free space are created. Machine language routines or data can then be safely POKEd into this area.

3. A better alternative is to use the Editor Assembler to assemble your subroutines. Then your subroutines can be linked directly to the compiled program and referenced using the CALL statement (see discussion of "CALL," section 9.2.1).

D.3 CALL Statement

You can also call assembly language subroutine's with the CALL statement. The syntax is:

CALL <global name> [(<argument list>)]

where <global name> contains an address that is the starting point in memory of the subroutine. <global name> cannot be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> can contain only variables.

A subroutine CALL with arguments results in a more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing parameters depends on the number of parameters to pass:

1. If the number of parameters is three or fewer, they are passed in the registers. Parameter 1 is in HL, 2 in DE (if present), and 3 in BC (if present).
2. If the number of parameters is greater than three, they are passed as follows:

   Parameter 1 in HL.

   Parameter 2 in DE.

   Parameters 3 through n in a contiguous data block. BC points to the low byte of this data block (For example, to the low byte of parameter 3).

Note: With this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than three parameters and needs to transfer them to a local data area, a system subroutine named $AT will perform this transfer. The $AT routine is listed below (it is located in the FORTRAN library, FORLIB/REL). The routine is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2).

The subroutine is responsible for saving the first two parameters before calling $AT. For example, if a subroutine expects 5 parameters, it should look like this:

```
SUBR:   LD      (P1),HL   ;SAVE PARAMETER 1
        EX      HL,DE
        LD      (P2),HL   ;SAVE PARAMETER 2
        LD      A,3       ;NO. OF PARAMETERS LEFT
        LD      HL,P3     ;POINTER TO LOCAL AREA
        CALL    $AT**     ;TRANSFER THE OTHER 3
                                PARAMETERS
        .
        .
        .
        [Body of subroutine]
        .
        .
        .
        RET               ;RETURN TO CALLER
P1:     DS      2         ;SPACE FOR PARAMETER 1
P2:     DS      2         ;SPACE FOR PARAMETER 2
P3:     DS      6         ;SPACE FOR PARAMETERS 3-5
```

The argument transfer routine $AT is:

```
;       ARGUMENT TRANSFER
;BC     POINTS TO 3RD PARAM.
;HL     POINTS TO LOCAL STORAGE FOR PARAM 3
;A      CONTAINS THE * OF PARAMS TO XFER (TOTAL-2)



$AT::   EX    HL,DE   ;SAVE HL IN DE
        LD    H,B
        LD    L,C     ;HL = PTR TO PARAMS
AT1:    LD    C,(HL)
        INC   HL
        LD    B,(HL)
        INC   HL      ;BC = PARAM ADR
        EX    HL,DE   ;HL POINTS TO LOCAL
                        STORAGE
        LD    (HL),C
        INC   HL
        LD    (HL),B
        INC   HL      ;STORE PARAM IN LOCAL AREA
        EX    HL,DE   ;SINCE GOING BACK TO AT1
        DEC   A       ;TRANSFERRED ALL PARAMS?
        JR    NZ,AT1  ;NO, COPY MORE
        RET           ;YES, RETURN
```

When accessing parameters in a subroutine, remember that they are **pointers** to the actual arguments passed.

Note: The programmer must match the **number**, **type**, and **length** of the arguments in the calling program with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those written in assembly language.

APPENDIX E

Disk File Handling

Two types of disk data files may be created and accessed by a BASIC program: sequential files and random access files.

E.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is a series of ASCII characters stored, one after another (sequentially), in the order it is sent. It is read back in the same way.

The statements and functions that are used with sequential files are:

OPEN
PRINT#
PRINT# USING
WRITE#
INPUT#
LINE INPUT#
EOF
LOC
CLOSE

See the *BASIC Reference Manual* for a more detailed discussion of these commands.

E.1.1. Creating a Sequential File

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.	OPEN "O",#1,"DATA"
2. Write data to the file using the WRITE# statement. (PRINT# may be used instead, but consult the *BASIC Reference Manual* before doing so.)	WRITE#1,A$;B$;C$
3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	CLOSE #1 OPEN "I",#1,"DATA"
4. Use the INPUT# statement to read data from the sequential file into the program.	INPUT#1,X$,Y$,Z$

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
    PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. For example,

```
100 IF LOC(1)>50 THEN STOP
```

would end program execution if more than 50 sectors had been written to or read from file #1 since it was OPENed.

Program 1 is a short program that creates a sequential file, named "DATA", from information you input at the terminal:

Program 1--Create a Sequential Data File

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
```

Execution of the program with sample input yields the following example:

```
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

Program 2 accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978:

Program 2--Accessing a Sequential File

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
```

Running the program gives:

```
EBENEEZER SCROOGE
SUPER MANN
Input past end at address xxxx
Ok
```

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 2Ø causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 4Ø to GOTO 15.

E.1.2 Adding Data to a Sequential File

Data can be appended to an existing sequential access file. It is important, however, to follow carefully the procedure given below.

Warning: If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents.

The following procedure can be used to add data to an existing file called "NAMES":

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY."
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY."
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
                 Program 3-- Adding Data
                   to a Sequential File

10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 N$="" :INPUT "NAME";N$
```

```
130 IF N$="" THEN 200: 'CARRIAGE RETURN
    EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",
     #2,"COPY":RESUME 120
2010 ON ERROR GOTO 10
```

The error handling routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

E.2 Random Access Files

Creating and accessing random files requires more program steps than sequential files. However, there are advantages to using random files, one of which is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk — it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN
FIELD
LSET/RSET
GET
PUT
CLOSE
LOC
MKD$
MKI$
MKS$
CVD
CVI
CVS

See the *BASIC Reference Manual* for detailed discussion of these statements and functions.

E.2.1 Creating a Random Access File

The following program steps are required to create a random access file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.	OPEN "R",#1,"FILE",32
2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.	FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.	LSET N$=X$ LSET A$=MKS$ (AMT) LSET P$=TEL$
4. Write the data from the buffer to the disk using the PUT statement.	PUT #1,CODE%

The LOC function with random files returns the "current record number." For example, the statement

    IF LOC (1)>50 THEN END

ends program execution if the current record number in file #1 is higher than 50.

Program 4 writes information that is input at the terminal to a random file.

Program 4--Create a Random Access File

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$ (AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

E.2.2 Accessing a Random Access File

The following program steps are required to access a random file:

1. OPEN the file in "R" mode.	OPEN "R",#1, "FILE",32
2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.	FIELD #1 20 AS N$, 4 AS A$, 8 AS P$

Note: In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.	GET #1, CODE%
4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.	PRINT N$ PRINT CVS(A$)

Program 5 accesses the random file "FILE" that was created in Program 4. When the two-digit code set up in Program 4 is input, the information associated with that code is read from the file and displayed:

```
          Program 5--Access a Random Access File

     10 OPEN "R",#1,"FILE",32
     20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
     30 INPUT "2-DIGIT CODE";CODE%
     40 GET #1,CODE%
     50 PRINT N$
     60 PRINT USING "$$###.##";CVS(A$)
     70 PRINT P$:PRINT
     80 GOTO 30
```

Program 6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number. It is assumed the inventory will contain no more than 100 different part numbers.

Note: This example must be compiled with the "-5" option switch. Press <Break> to stop program.

Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
                    Program 6--Inventory

     120 OPEN"R",#1,"INVEN/DAT",39
     125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,
                   2 AS R$,4 AS P$
     130 PRINT:PRINT "FUNCTIONS:":PRINT
     135 PRINT 1,"INITIALIZE FILE"
     140 PRINT 2,"CREATE A NEW ENTRY"
```

```
150 PRINT 3,"DISPLAY INVENTORY
            FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW
            REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6)
            THEN PRINT
            "BAD FUNCTION NUMBER":GOTO 130
230 ON FUNCTION GOSUB
            900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";
            A$: IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT
            "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";
            CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";
            CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";
            CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT
            "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT
            "NULL ENTRY":RETURN
```

```
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT":S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;"
            IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT
            "QUANTITY NOW";Q%;
            " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;
            " QUANTITY"; CVI(Q$) TAB(50)
            "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN
            PRINT "BAD PART NUMBER":
            GOTO 840
            ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:
            IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

APPENDIX F

Floating-Point Numeric Format

This discussion provides the information needed to encode and decode the floating-point representation. This information is intended for advanced assembly language programmers, and should not be viewed as an introduction to binary math.

Note that the encoding information presented below pertains only to integral numbers. Encoding fractional numbers is a very complex process.

F.1 Encoding an Integral Floating-Point Number

The floating-point representation is a normalized binary approximation of the argument number. It consists of two parts, the mantissa and the exponent.

The mantissa is a 24-bit (single precision) or 56-bit (double precision) normalized approximation of the number. The most significant bit of the mantissa is always assumed to be a 1, after normalization. Therefore, this bit is free to represent the sign of the mantissa.

The exponent is an "excess-8Ø" (8ØH) representation of the binary (powers of two) exponent of the number. 8ØH is added to the binary exponent, so that positive exponents are assumed to have an exponent of 8ØH or greater, while negative exponents are assumed to have an exponent of 7FH or less. An exponent of zero indicates the number itself is zero, regardless of the mantissa.

The procedure for encoding an integral number into floating-point representation consists of 4 steps:

1. convert to binary
2. normalize
3. compute the exponent
4. store

This process may best be explained by example. In the steps explained below, the number 5.ØØ is converted to a single precision number.

1. The conversion to binary may be done in many ways. The simplest of these is the subtraction method.

   This method uses repeated subtractions of the powers of two until the number is converted. For the purposes of our example, a partial table of the positive powers of two is shown:

   $2^Ø=1$ $2^1=2$ $2^2=4$ $2^3=8$ $2^4=16$, . . .

   Subtract the largest power of two that produces a positive result or zero. If the result is positive or zero, mark a 1 in the binary equivalent column as shown below. If the result is a negative number, mark a zero in the binary equivalent column. If there is a remainder, repeat the subtraction process with the next power of two.

For example:

Conversion		Binary equivalent
5-4=1	—	1

4 (2^2) is the largest number that can be subtracted from 5. The result is a remainder of 1.

Now, see if the next power of two (2^1) can be subtracted from the remainder.

Conversion		Binary equivalent
1-2=-1	—	01

Since 1-2 produces a negative number, do not subtract. Instead, mark a zero.

Repeat the subtraction process with the next largest power of two ($2^0=1$).

Conversion		Binary equivalent
1-1=0	—	101

One will subtract evenly, so the final binary result is 101.

Note: If you get to the point of subtracting 1 and the result is not zero, you have made an error.

2. Now the binary number must be normalized. This is accomplished by moving the **binary point** (the binary equivalent of the decimal point) to the left until it is immediately left of the leftmost 1 of the number (the Most Significant Bit); as the point is moved, count the number of "shifts" that were made. Thus, 101.00 . . . becomes .10100 . . .

   The next step in normalization is converting the Most Significant Bit into the sign bit. Because floating-point representation assumes that the Most Significant Bit is 1 (this is why the number is normalized), this bit represents the sign of the number. Since the original number was positive, the sign bit becomes zero (1 indicates negative). Therefore, the normalized number is .0010 0000 . . .

3. To convert the number to its final form, calculate the exponent by adding 80H to the number of shifts performed during normalization. Since the binary point was shifted 3 places, add 3. This results in an exponent of 83H. The floating-point number is therefore .0010 0000 0000 0000 0000 0000 with an exponent of 83H, or 00 00 20 83 in Hex.

4. The floating-point number is stored as LSB (Least Significant Byte), NSB (Next Significant Byte), MSB (Most Significant Byte), and EXP (Exponent), with LSB at low memory and EXP at high memory. This is the form presented by a USR function call or a CALL statement.

F.2 Decoding an Integral Floating-Point Number

To decode an integral floating-point number, perform the above steps in reverse: Find the MSB of the mantissa, check the Most Significant Bit for sign, set the MSB, and de-normalize. For example, the following steps are required to decode 00 00 20 83 Hex:

1. Check the Most Significant Bit of the Most Significant Byte (MSB) for the sign of the number. In this case, the MSB is 0010 0000, so the sign of the number is positive.
2. Set the Most Significant Bit to 1. This results in a binary number of 1010 0000.
3. De-normalize the number by shifting the binary point the necessary number of places. 83H implies shifting the binary point 3 places right, giving us 101.0 0000, or 5 decimal.

F.3 Decoding a Fractional Floating-Point Number

If the number to be converted is a fraction, it will have a negative exponent.

A negative exponent (7F or less) simply implies that the binary point is shifted to the left instead of the right when decoding. Therefore 1010 0000 with an exponent of 7DH would become .0001 0100 after de-normalization. Because the sign bit was set, we know the original number was negative. Computing from the negative powers of two, we have $2^{-4}+2^{-6} = .0625+.015625 =$ .078125. Since the sign of the number is negative, the final result is -.078125.

Error Messages

During development of a BASIC program with the BASIC Compiler, four different kinds of errors may occur: BASIC Compiler severe errors, BASIC Compiler warning errors, L80 errors, and BASIC runtime errors. This chapter lists error codes, error numbers, and error messages.

G.1 Compiletime Errors

For errors that occur at compile time, the compiler outputs the line containing the error, an arrow beneath that line pointing to the place in the line where the error occurred, and a two-character code for the error. In some cases, the compiler reads ahead on a line to determine whether an error has actually occurred. In those cases, the arrow points a few characters beyond the error, or to the end of the line.

The BASIC Compiletime errors described below are divided into Severe Errors and Warning Errors.

Severe Errors

CODE	MESSAGE
BS	Bad Subscript Illegal dimension value Wrong number of subscripts
CD	Duplicate COMMON variable
CN	COMMON array not dimensioned
CO	COMMON out of order
DD	Array Already Dimensioned
FD	Function Already Defined
FN	FOR/NEXT Error FOR loop index variable already in use FOR without NEXT NEXT without FOR
IN	INCLUDE Error $INCLUDE file not found
LL	Line Too Long
LS	String Constant Too Long
OM	Out of Memory Array too big Data memory overflow Too many statement numbers Program memory overflow
OV	Math Overflow
SN	Syntax error — caused by one of the following: Illegal argument name Illegal assignment target Illegal constant format

APPENDIX G

	Illegal debug request
	Illegal DEFxxx character specification
	Illegal expression syntax
	Illegal function argument list
	Illegal function name
	Illegal function formal parameter
	Illegal separator
	Illegal format for statement number
	Illegal subroutine syntax
	Invalid character
	Missing AS
	Missing equal sign
	Missing GOTO or GOSUB
	Missing comma
	Missing INPUT
	Missing line number
	Missing left parenthesis
	Missing minus sign
	Missing operand in expression
	Missing right parenthesis
	Missing semicolon
	Missing slash
	Name too long
	Expected GOTO or GOSUB
	String assignment required
	String expression required
	String variable required
	Illegal syntax
	Variable required
	Wrong number of arguments
	Formal parameters must be unique
	Single variable only allowed
	Missing TO
	Illegal FOR loop index variable
	Illegal COMMON name
	Missing THEN
	Missing BASE
	Illegal subroutine name
SQ	Sequence Error
	Duplicate statement number
	Statement out of sequence
TC	Too Complex
	Expression too complex
	Too many arguments in function call
	Too many dimensions
	Too many variables for LINE INPUT
	Too many variables for INPUT
TM	Type Mismatch
	Data type conflict
	Variable must be of same type

UC	Unrecognizable Command Statement unrecognizable Command not implemented
UF	Function Not Defined
WE	WHILE/WEND Error WHILE without WEND WEND without WHILE
/0	Division by Zero
-E	Missing "-E" Switch
-X	Missing "-X" Switch

Warning Errors

CODE	MESSAGE
MC	Metacommand Error
ND	Array not Dimensioned
SI	Statement Ignored Statement ignored Unimplemented command

G.2 L80 Errors

?Loading Error	The last file given for input was not a properly formatted L80 object file.
?Out of Memory	Not enough memory to load program.
?Command Error	Unrecognizable L80 command.
?<file> Not Found	<file>, as given in the command string, did not exist.

%Mult. Def. Global YYYYYY

More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process. This means two subroutines with the same ENTRY point were specified in the L80 command line.

%Overlaying Program Area
Data

?Intersecting Program Area
Data

If you receive either of these error messages, you have set the -D and -P switches too close together. Reset the :<address> portion of both switches so that the locations are farther apart.

Origin Above Loader Memory, Move Anyway (Y or N)?
Below

Loader memory is 5200H to high memory. If you received this error message, you specified the -D or the -P switch with an address outside this range. Reset the :<address> portion of the switch(es).

?Can't Save Object File

A disk error occurred when the file was being saved. Almost always when you receive this message, you can assume that there is not enough disk space free in which to store the program.

G.3 Runtime Errors

The following errors may occur at program runtime. The error numbers match those issued by the BASIC Interpreter. The compiler runtime system prints long error messages followed by an address, unless -D, -E, or -X is specified in the compiler command line. In those cases, the error message is also followed by the number of the line in which the error occurred.

When you receive an error that gives the address where the error occurred, you can review the listing file to find the correct line number. Subtract the hexadecimal address of the program origin from the error address. Find the difference in the left column of hexadecimal numbers in the listing file. When you find the line which corresponds to the difference between the program origin and the error address, the line number will be the line which contains the error.

Note that if you have deleted the listing file, you will need to recompile it. Enter the command:

```
BASCOM ,LSTfile:<drive>=BASfile
```

Or, if you have a printer, obtain a hard copy:

```
BASCOM ,LPT=BASfile
```

The error numbers given below correspond to the value returned for BASIC.

NUMBER MESSAGE

2 Syntax Error
A line is encountered that contains an incorrect sequence of characters in a DATA statement.

3 RETURN without GOSUB
A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.

4 Out of Data
A READ statement is executed when there are no DATA statements with unread data remaining in the program.

5	Illegal Function Call A parameter that is out of range is passed to a math or string function. A function call error may also occur as the result of: A negative or unreasonably large subscript A negative or zero argument with LOG A negative argument to SQR A negative mantissa with a non-integer exponent A call to a USR function for which the starting address has not yet been given An improper argument to ASC, CHR$, MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON . . . GOTO A string concatenation that is longer than 255 characters
6	Floating Overflow or Integer Overflow The result of a calculation is too large to be represented within the range allowed for floating point numbers.
9	Subscript Out of Range An array element is referenced with a subscript that is outside the dimensions of the array.
11	Division by Zero A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power.
14	Out of String Space String variables exceed the allocated amount of string space.
19	RESUME without Error A RESUME statement is encountered before an error trapping routine is entered.
21	Unprintable Error An error message is not available for the error condition that exists. This is usually caused by an ERROR with an undefined error code.
5Ø	Field Overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal Error An internal malfunction occurs in the BASIC Compiler.

52	Bad File Number	A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
53	File Not Found	A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.
54	Bad File Mode	An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN with a file mode other than I, O, or R.
55	File Already Open	A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.
57	Disk I/O Error	An I/O error occurred on a disk I/O operation. The operating system cannot recover from the error.
58	File Already Exists	The filename specified in a NAME statement is identical to a filename already in use on the disk.
61	Disk Full	All disk space has been allocated.
62	Input Past End	An INPUT statement reads from a null (empty) file, or from a file in which all data has already been read. To avoid this error, use the EOF function to detect the end of file.
63	Bad Record Number	In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or is equal to Ø.
64	Bad File Name	An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).
67	Too Many Files	The 255 file directory maximum is exceeded by an attempt to create a new file with SAVE or OPEN.

The following additional runtime error messages are severe and cannot be trapped:

Internal Error — String Space Corrupt

Internal Error — String Space Corrupt during G.C.

Internal Error — No Line Number

The first two errors usually occur because a string descriptor has been improperly modified. (G.C. stands for garbage collection.)

The last error occurs when the error address cannot be found in the line number table during error trapping. This occurs if you have forgotten to use either the -X or -E compiler switch for programs that use RESUME and ON ERROR GOTO statements.

INDEX